

Let's Go!

LET'S GO *FURTHER*

ADVANCED PATTERNS FOR BUILDING, MANAGING AND
DEPLOYING RESTFUL **JSON APIS AND WEB APPLICATIONS**
IN GO



ALEX EDWARDS

1ST EDITION

Contents

- **1. Introduction**
 - 1.1. Prerequisites
- **2. Getting Started**
 - 2.1. Project Setup and Skeleton Structure
 - 2.2. A Basic HTTP Server
 - 2.3. API Endpoints and RESTful Routing
- **3. Sending JSON Responses**
 - 3.1. Fixed-Format JSON
 - 3.2. JSON Encoding
 - 3.3. Encoding Structs
 - 3.4. Formatting and Enveloping Responses
 - 3.5. Advanced JSON Customization
 - 3.6. Sending Error Messages
- **4. Parsing JSON Requests**
 - 4.1. JSON Decoding
 - 4.2. Managing Bad Requests
 - 4.3. Restricting Inputs
 - 4.4. Custom JSON Decoding
 - 4.5. Validating JSON Input
- **5. Database Setup and Configuration**
 - 5.1. Setting up PostgreSQL
 - 5.2. Connecting to PostgreSQL
 - 5.3. Configuring the Database Connection Pool
- **6. SQL Migrations**
 - 6.1. An Overview of SQL Migrations
 - 6.2. Working with SQL Migrations
- **7. CRUD Operations**
 - 7.1. Setting up the Movie Model
 - 7.2. Creating a New Movie
 - 7.3. Fetching a Movie
 - 7.4. Updating a Movie

- 7.5. Deleting a Movie
- **8. Advanced CRUD Operations**
 - 8.1. Handling Partial Updates
 - 8.2. Optimistic Concurrency Control
 - 8.3. Managing SQL Query Timeouts
- **9. Filtering, Sorting, and Pagination**
 - 9.1. Parsing Query String Parameters
 - 9.2. Validating Query String Parameters
 - 9.3. Listing Data
 - 9.4. Filtering Lists
 - 9.5. Full-Text Search
 - 9.6. Sorting Lists
 - 9.7. Paginating Lists
 - 9.8. Returning Pagination Metadata
- **10. Structured Logging and Error Handling**
 - 10.1. Structured JSON Log Entries
 - 10.2. Panic Recovery
- **11. Rate Limiting**
 - 11.1. Global Rate Limiting
 - 11.2. IP-based Rate Limiting
 - 11.3. Configuring the Rate Limiters
- **12. Graceful Shutdown**
 - 12.1. Sending Shutdown Signals
 - 12.2. Intercepting Shutdown Signals
 - 12.3. Executing the Shutdown
- **13. User Model Setup and Registration**
 - 13.1. Setting up the Users Database Table
 - 13.2. Setting up the Users Model
 - 13.3. Registering a User
- **14. Sending Emails**
 - 14.1. SMTP Server Setup
 - 14.2. Creating Email Templates
 - 14.3. Sending a Welcome Email
 - 14.4. Sending Background Emails

- 14.5. Graceful Shutdown of Background Tasks
- **15. User Activation**
 - 15.1. Setting up the Tokens Database Table
 - 15.2. Creating Secure Activation Tokens
 - 15.3. Sending Activation Tokens
 - 15.4. Activating a User
- **16. Authentication**
 - 16.1. Authentication Options
 - 16.2. Generating Authentication Tokens
 - 16.3. Authenticating Requests
- **17. Permission-based Authorization**
 - 17.1. Requiring User Activation
 - 17.2. Setting up the Permissions Database Table
 - 17.3. Setting up the Permissions Model
 - 17.4. Checking Permissions
 - 17.5. Granting Permissions
- **18. Cross Origin Requests**
 - 18.1. An Overview of CORS
 - 18.2. Demonstrating the Same-Origin Policy
 - 18.3. Simple CORS Requests
 - 18.4. Preflight CORS Requests
- **19. Metrics**
 - 19.1. Exposing Metrics with Expvar
 - 19.2. Creating Custom Metrics
 - 19.3. Request-level Metrics
 - 19.4. Recording HTTP Status Codes
- **20. Building, Versioning and Quality Control**
 - 20.1. Creating and Using Makefiles
 - 20.2. Managing Environment Variables
 - 20.3. Quality Controlling Code
 - 20.4. Module Proxies and Vendoring
 - 20.5. Building Binaries
 - 20.6. Managing and Automating Version Numbers
- **21. Deployment and Hosting**

- [21.1. Creating a Digital Ocean Droplet](#)
- [21.2. Server Configuration and Installing Software](#)
- [21.3. Deployment and Executing Migrations](#)
- [21.4. Running the API as a Background Service](#)
- [21.5. Using Caddy as a Reverse Proxy](#)
- **22. Appendices**
 - [22.1. Managing Password Resets](#)
 - [22.2. Creating Additional Activation Tokens](#)
 - [22.3. Authentication with JSON Web Tokens](#)
 - [22.4. JSON Encoding Nuances](#)
 - [22.5. JSON Decoding Nuances](#)
 - [22.6. Request Context Timeouts](#)
- **23. Feedback**

Introduction

In this book we're going to work through the start-to-finish build of an application called *Greenlight* — a JSON API for retrieving and managing information about movies. You can think of the core functionality as being a bit like the [Open Movie Database API](#).

Ultimately, our *Greenlight* API will support the following endpoints and actions:

Method	URL Pattern	Action
GET	/v1/healthcheck	Show application health and version information
GET	/v1/movies	Show the details of all movies
POST	/v1/movies	Create a new movie
GET	/v1/movies/:id	Show the details of a specific movie
PATCH	/v1/movies/:id	Update the details of a specific movie
DELETE	/v1/movies/:id	Delete a specific movie
POST	/v1/users	Register a new user
PUT	/v1/users/activated	Activate a specific user
PUT	/v1/users/password	Update the password for a specific user
POST	/v1/tokens/authentication	Generate a new authentication token
POST	/v1/tokens/password-reset	Generate a new password-reset token
GET	/debug/vars	Display application metrics

To give you an idea of what the API will look like from a client's point of view, by the end of this book the `GET /v1/movies/:id` endpoint will return a response similar this:

```
$ curl -H "Authorization: Bearer RIDBIAE3AMMK57T6IAEBUGA7ZQ" localhost:4000/v1/movies/1
{
  "movie": {
    "id": 1,
    "title": "Moana",
    "year": 2016,
    "runtime": "107 mins",
    "genres": [
      "animation",
      "adventure"
    ],
    "version": 1
  }
}
```

Behind the scenes, we'll use PostgreSQL as the database for persistently storing all the data. And at the end of the book, we'll deploy the finished API to a Linux server running on Digital Ocean.

Conventions

In this book, code blocks are shown with a silver background like the snippet below. If the code block is particularly long, parts that aren't relevant may be replaced with an ellipsis. To make it easy to follow along, most code blocks also have a title bar at the top indicating the name of the file that we're working on.

File: hello.go

```
package main

... // Indicates that some existing code has been omitted.

func sayHello() {
    fmt.Println("Hello world!")
}
```

Terminal (command line) instructions are shown with a black background and generally start with a dollar symbol. These commands should work on any Unix-based operating system, including macOS and Linux. Sample output is shown in silver beneath the command, like so:

```
$ echo "Hello world!"
Hello world!
```

If you're using Windows, you should replace these commands with the DOS equivalent or carry out the action via the normal Windows GUI.

Some chapters in this book end with an *additional information* section. These sections contain information that isn't relevant to our application build, but is still important (or sometimes, just interesting) to know about.

About the author

Hey, I'm Alex Edwards, a full-stack web developer and author. I live near Innsbruck, Austria.

I've been working with Go for over 8 years, building production applications for myself and commercial clients, and helping people all around the world improve their Go skills.

You can see more of my writing on [my blog](#) (where I publish detailed tutorials), some of my open-source work on [GitHub](#), and you can also follow me on [Instagram](#) and [Twitter](#).

Copyright and disclaimer

Let's Go Further. Copyright © 2021 Alex Edwards.

Last updated 2021-08-27 08:41:48 UTC. Version 1.1.0.

The Go gopher was designed by [Renee French](#) and is used under the Creative Commons 3.0 Attributions license. Cover gopher adapted from vectors by [Egon Elbre](#).

The information provided within this book is for general informational purposes only. While the author and publisher have made every effort to ensure that the accuracy of the information within this book was correct at time of publication there are no representations or warranties, express or implied, about the completeness, accuracy, reliability, suitability or availability with respect to the information, products, services, or related graphics contained in this book for any purpose. Any use of this information is at your own risk.

Prerequisites

Background knowledge

This book is written as a follow up to [Let's Go](#), and we'll leverage a lot of the information and code patterns from that book again here.

If you've already read and enjoyed *Let's Go*, then this book should be a good fit for you and the ideal next step in your learning. If you haven't, then I highly recommend starting with *Let's Go* first — especially if you're a newcomer to Go.

You can read this as a standalone book, but please be aware that it is somewhat advanced — it doesn't explain the fundamentals in detail, and some topics (like testing) don't feature at all because they were covered heavily in the previous book. But if you're comfortable using Go and already have a decent amount of experience behind you, then this book may also be a good fit for you. Feel free to jump straight in.

Go 1.17

The information in this book is correct for the [latest major release of Go](#) (version 1.17), and you should install this if you'd like to code-along with the application build.

If you've already got Go installed, you can check the version number from your terminal by using the `go version` command. The output should look similar to this:

```
$ go version
go version go1.17 linux/amd64
```

If you need to upgrade your version of Go, then please go ahead and do that now. The instructions for your operating system [can be found here](#).

Other software

There are a few other bits of software that you should make sure are available on your computer if you want to follow along fully. They are:

- The [curl](#) tool for working with HTTP requests and responses from your terminal. On MacOS and Linux machines it should be pre-installed or available in your software repositories. Otherwise, you can download the latest version [from here](#).
- The [hey](#) tool for carrying out some basic load tests. So long as you have Go 1.16 or newer on your computer, you can install hey with the `go install` command:

```
$ go install github.com/rakyll/hey@latest
```

- The [git](#) version control system. Installation instructions for all operating systems can be [found here](#).
- A web browser with good developer tools. I'll be using [Firefox](#) in this book, but Chromium, Chrome or Microsoft Edge will work too.
- Your favorite text editor 😊

Getting Started

In this first section of the book, we're going to set up a project directory and lay the groundwork for building our *Greenlight* API. We will:

- Create a skeleton directory structure for the project and explain at a high-level how our Go code and other assets will be organized.
- Establish a HTTP server to listen for incoming HTTP requests.
- Introduce a sensible pattern for managing configuration settings (via command-line flags) and using *dependency injection* to make dependencies available to our handlers.
- Use the [httprouter](#) package to help implement a standard RESTful structure for the API endpoints.

Project Setup and Skeleton Structure

Let's kick things off by creating a **greenlight** directory to act as the top-level 'home' for this project. I'm going to create my project directory at `$HOME/Projects/greenlight`, but feel free to choose a different location if you wish.

```
$ mkdir -p $HOME/Projects/greenlight
```

Then change into this directory and use the **go mod init** command to *enable modules* for the project.

When running this command you'll need to specify a *module path*, which is essentially a unique identifier for your project. In this book I'll use **greenlight.alexedwards.net** as my module path, but if you're following along you should ideally swap this for something that is unique to you instead.

```
$ cd $HOME/Projects/greenlight
$ go mod init greenlight.alexedwards.net
go: creating new go.mod: module greenlight.alexedwards.net
```

At this point you'll see that a **go.mod** file has been created in the root of your project directory. If you open it up, it should look similar to this:

```
File: go.mod

module greenlight.alexedwards.net

go 1.17
```

We talked about modules in detail as part of the first *Let's Go* book, but as a quick refresher let's recap the main points here.

- When there is a valid **go.mod** file in the root of your project directory, your project *is a module*.
- When you're working inside your project directory and download a dependency with **go get**, then the exact version of the dependency will be recorded in the **go.mod** file. Because the exact version is known, this makes it much easier to ensure reproducible builds across different machines and environments.

- When you run or build the code in your project, Go will use the exact dependencies listed in the `go.mod` file. If the necessary dependencies aren't already on your local machine, then Go will automatically download them for you — along with any recursive dependencies too.
- The `go.mod` file also defines the *module path* (which is `greenlight.alexedwards.net` in my case). This is essentially the identifier that will be used as the *root import path* for the packages in your project.
- It's good practice to make the module path unique to you and your project. A common convention in the Go community is to base it on a URL that you own.

Hint: If you feel unsure about any aspect of modules or how they work, the official [Go Modules Wiki](#) is an excellent resource and contains answers to a wide range of [FAQs](#) — although please be aware that it hasn't yet been fully updated for Go 1.17 at the time of writing.

Generating the skeleton directory structure

Alright, now that our project directory has been created and we have a `go.mod` file, you can go ahead and run the following commands to generate a high-level skeleton structure for the project:

```
$ mkdir -p bin cmd/api internal migrations remote
$ touch Makefile
$ touch cmd/api/main.go
```

At this point your project directory should look exactly like this:

```
.
├── bin
├── cmd
│   └── api
│       └── main.go
├── internal
├── migrations
├── remote
├── go.mod
└── Makefile
```

Let's take a moment to talk through these files and folders and explain the purpose that they'll serve in our finished project.

- The `bin` directory will contain our compiled application binaries, ready for deployment to a production server.
- The `cmd/api` directory will contain the application-specific code for our *Greenlight* API application. This will include the code for running the server, reading and writing HTTP requests, and managing authentication.
- The `internal` directory will contain various ancillary packages used by our API. It will contain the code for interacting with our database, doing data validation, sending emails and so on. Basically, any code which *isn't* application-specific and can potentially be reused will live in here. Our Go code under `cmd/api` will *import* the packages in the `internal` directory (but never the other way around).
- The `migrations` directory will contain the SQL migration files for our database.
- The `remote` directory will contain the configuration files and setup scripts for our production server.
- The `go.mod` file will declare our project dependencies, versions and module path.
- The `Makefile` will contain *recipes* for automating common administrative tasks — like auditing our Go code, building binaries, and executing database migrations.

It's important to point out that the directory name `internal` carries a special meaning and behavior in Go: any packages which live under this directory can only be imported by code *inside the parent of the `internal` directory*. In our case, this means that any packages which live in `internal` can only be imported by code inside our `greenlight` project directory.

Or, looking at it the other way, this means that any packages under `internal` *cannot be imported by code outside of our project*.

This is useful because it prevents other codebases from importing and relying on the (potentially unversioned and unsupported) packages in our `internal` directory — even if the project code is publicly available somewhere like GitHub.

Hello world!

Before we continue, let's quickly check that everything is setup correctly. Open the `cmd/api/main.go` file in your text editor and add the following code:

File: cmd/api/main.go

```
package main

import "fmt"

func main() {
    fmt.Println("Hello world!")
}
```

Save this file, then use the `go run` command in your terminal to compile and execute the code in the `cmd/api` package. All being well, you will see the following output:

```
$ go run ./cmd/api
Hello world!
```

A Basic HTTP Server

Now that the skeleton structure for our project is in place, let's focus our attention on getting a HTTP server up and running.

To start with, we'll configure our server to have just one endpoint: `/v1/healthcheck`. This endpoint will return some basic information about our API, including its current version number and operating environment (development, staging, production, etc.).

URL Pattern	Handler	Action
<code>/v1/healthcheck</code>	<code>healthcheckHandler</code>	Show application information

If you're following along, open up the `cmd/api/main.go` file and replace the 'hello world' application with the following code:

File: main.go

```
package main

import (
    "flag"
    "fmt"
    "log"
    "net/http"
    "os"
    "time"
)

// Declare a string containing the application version number. Later in the book we'll
// generate this automatically at build time, but for now we'll just store the version
// number as a hard-coded global constant.
const version = "1.0.0"

// Define a config struct to hold all the configuration settings for our application.
// For now, the only configuration settings will be the network port that we want the
// server to listen on, and the name of the current operating environment for the
// application (development, staging, production, etc.). We will read in these
// configuration settings from command-line flags when the application starts.
type config struct {
    port int
    env string
}

// Define an application struct to hold the dependencies for our HTTP handlers, helpers,
// and middleware. At the moment this only contains a copy of the config struct and a
// logger, but it will grow to include a lot more as our build progresses.
type application struct {
    config config
    logger *log.Logger
}
```



```

func main() {
    // Declare an instance of the config struct.
    var cfg config

    // Read the value of the port and env command-line flags into the config struct. We
    // default to using the port number 4000 and the environment "development" if no
    // corresponding flags are provided.
    flag.IntVar(&cfg.port, "port", 4000, "API server port")
    flag.StringVar(&cfg.env, "env", "development", "Environment (development|staging|production)")
    flag.Parse()

    // Initialize a new logger which writes messages to the standard out stream,
    // prefixed with the current date and time.
    logger := log.New(os.Stdout, "", log.Ldate | log.Ltime)

    // Declare an instance of the application struct, containing the config struct and
    // the logger.
    app := &application{
        config: cfg,
        logger: logger,
    }

    // Declare a new servemux and add a /v1/healthcheck route which dispatches requests
    // to the healthcheckHandler method (which we will create in a moment).
    mux := http.NewServeMux()
    mux.HandleFunc("/v1/healthcheck", app.healthcheckHandler)

    // Declare a HTTP server with some sensible timeout settings, which listens on the
    // port provided in the config struct and uses the servemux we created above as the
    // handler.
    srv := &http.Server{
        Addr:      fmt.Sprintf(":%d", cfg.port),
        Handler:    mux,
        IdleTimeout: time.Minute,
        ReadTimeout: 10 * time.Second,
        WriteTimeout: 30 * time.Second,
    }

    // Start the HTTP server.
    logger.Printf("starting %s server on %s", cfg.env, srv.Addr)
    err := srv.ListenAndServe()
    logger.Fatal(err)
}

```

Important: If any of the terminology or patterns in the code above are unfamiliar to you, then I strongly recommend reading the first [Let's Go](#) book which explains them in detail.

Creating the healthcheck handler

The next thing we need to do is create the `healthcheckHandler` method for responding to HTTP requests. For now, we'll keep the logic in this handler really simple and have it return a plain-text response containing three pieces of information:

- A fixed `"status: available"` string.
- The API version from the hard-coded `version` constant.
- The operating environment name from the `env` command-line flag.

Go ahead and create a new `cmd/api/healthcheck.go` file:

```
$ touch cmd/api/healthcheck.go
```

And then add the following code:

```
File: cmd/api/healthcheck.go

package main

import (
    "fmt"
    "net/http"
)

// Declare a handler which writes a plain-text response with information about the
// application status, operating environment and version.
func (app *application) healthcheckHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "status: available")
    fmt.Fprintf(w, "environment: %s\n", app.config.env)
    fmt.Fprintf(w, "version: %s\n", version)
}
```

The important thing to point out here is that `healthcheckHandler` is implemented as a *method* on our `application` struct.

This is an effective and idiomatic way to make dependencies available to our handlers without resorting to global variables or closures — any dependency that the `healthcheckHandler` needs can simply be included as a field in the `application` struct when we initialize it in `main()`.

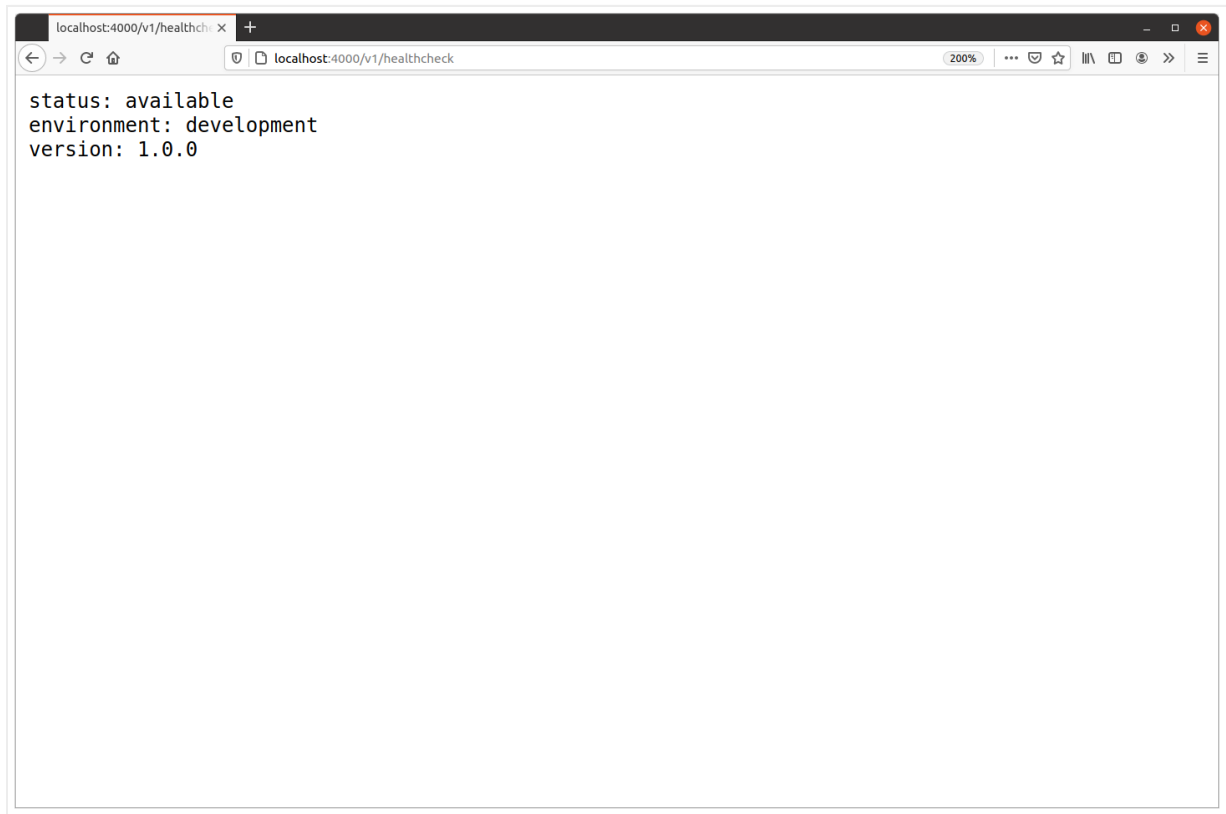
We can see this pattern already being used in the code above, where the operating environment name is retrieved from the `application` struct by calling `app.config.env`.

Demonstration

OK, let's try this out. Make sure that all your changes are saved, then use the `go run` command again to execute the code in the `cmd/api` package. You should see a log message confirming that the HTTP server is running, similar to this:

```
$ go run ./cmd/api
2021/04/05 19:42:50 starting development server on :4000
```

While the server is running, go ahead and try visiting localhost:4000/v1/healthcheck in your web browser. You should get a response from the `healthcheckHandler` which looks like this:



Or alternatively, you can use `curl` to make the request from your terminal:

```
$ curl -i localhost:4000/v1/healthcheck
HTTP/1.1 200 OK
Date: Mon, 05 Apr 2021 17:46:14 GMT
Content-Length: 58
Content-Type: text/plain; charset=utf-8

status: available
environment: development
version: 1.0.0
```

Note: The `-i` flag in the command above instructs `curl` to display the HTTP response headers as well as the response body.

If you want, you can also verify that the command-line flags are working correctly by

specifying alternative **port** and **env** values when starting the application. When you do this, you should see the contents of the log message change accordingly. For example:

```
$ go run ./cmd/api -port=3030 -env=production
2021/04/05 19:48:34 starting production server on :3030
```

Additional Information

API versioning

APIs which support real-world businesses and users often need to change their functionality and endpoints over time — sometimes in a backwards-incompatible way. So, to avoid problems and confusion for clients, it's a good idea to always implement some form of API *versioning*.

There are [two common approaches](#) to doing this:

1. By prefixing all URLs with your API version, like **/v1/healthcheck** or **/v2/healthcheck**.
2. By using custom **Accept** and **Content-Type** headers on requests and responses to convey the API version, like **Accept: application/vnd.greenlight-v1**.

From a HTTP semantics point of view, using headers to convey the API version is the ‘purer’ approach. But from a user-experience point of view, using a URL prefix is arguably better. It makes it possible for developers to see which version of the API is being used at a glance, and it also means that the API can still be explored using a regular web browser (which is harder if custom headers are required).

Throughout this book we'll version our API by prefixing all the URL paths with **/v1/** — just like we did with the **/v1/healthcheck** endpoint in this chapter.

API Endpoints and RESTful Routing

Over the next few sections of this book we're going to gradually build up our API so that the endpoints start to look like this:

Method	URL Pattern	Handler	Action
GET	/v1/healthcheck	healthcheckHandler	Show application information
GET	/v1/movies	listMoviesHandler	Show the details of all movies
POST	/v1/movies	createMovieHandler	Create a new movie
GET	/v1/movies/:id	showMovieHandler	Show the details of a specific movie
PUT	/v1/movies/:id	editMovieHandler	Update the details of a specific movie
DELETE	/v1/movies/:id	deleteMovieHandler	Delete a specific movie

If you've built APIs with [REST](#) style endpoints before, then the table above probably looks very familiar to you and doesn't require much explanation. But if you're new to this, then there are a couple of important things to point out.

The first thing is that requests with the same URL pattern will be routed to *different handlers based on the [HTTP request method](#)*. For both security and semantic correctness, it's important that we use the appropriate HTTP method for the action that the handler is performing.

In summary:

Method	Usage
GET	Use for actions that retrieve information only and don't change the state of your application or any data.
POST	Use for non-idempotent actions that modify state. In the context of a REST API, POST is generally used for actions that <i>create</i> a new resource.
PUT	Use for idempotent actions that modify the state of a resource at a specific URL. In the context of a REST API, PUT is generally used for actions that <i>replace</i> or <i>update</i> an existing resource.
PATCH	Use for actions that <i>partially update</i> a resource at a specific URL. It's OK for the action to be either idempotent or non-idempotent.
DELETE	Use for actions that <i>delete</i> a resource at a specific URL.

The other important thing to point out is that our API endpoints will use [clean URLs](#), with parameters interpolated in the URL path. So — for example — to retrieve the details of a specific movie a client will make a request like **GET** `/v1/movies/1`, instead of appending the movie ID in a query string parameter like **GET** `/v1/movies?id=1`.

Choosing a router

When you're building an API with endpoints like this in Go, one of the first hurdles you'll meet is the fact that [http.ServeMux](#) — the router in the Go standard library — is quite limited in terms of its functionality. In particular it doesn't allow you to route requests to different handlers based on the request method (**GET**, **POST**, etc.), nor does it provide support for clean URLs with interpolated parameters.

Although you can [work-around](#) these limitations, or [implement your own router](#), generally it's easier to use one of the many third-party routers that are available instead.

In this chapter we're going to integrate the popular [httprouter](#) package with our application. Most importantly, **httprouter** is stable, well-tested and provides the functionality we need — and as a bonus it's also [extremely fast](#) thanks to its use of a radix tree for URL matching. If you're building a REST API for public consumption, then **httprouter** is a solid choice.

If you're coding-along with this book, please use `go get` to download version **v1.3.0** of **httprouter** like so:

```
$ go get github.com/julienschmidt/httprouter@v1.3.0
go: downloading github.com/julienschmidt/httprouter v1.3.0
go get: added github.com/julienschmidt/httprouter v1.3.0
```

Note: If you already have a copy of **httprouter v1.3.0** on your machine from another project, then your existing copy will be used and you won't see a **go: downloading...** message.

To demonstrate how **httprouter** works, we'll start by adding the two endpoints for *creating a new movie* and *showing the details of a specific movie* to our codebase. By the end of this chapter, our API endpoints will look like this:

Method	URL Pattern	Handler	Action
GET	/v1/healthcheck	healthcheckHandler	Show application information
POST	/v1/movies	createMovieHandler	Create a new movie
GET	/v1/movies/:id	showMovieHandler	Show the details of a specific movie

Encapsulating the API routes

To prevent our **main()** function from becoming cluttered as the API grows, let's encapsulate all the routing rules in a new **cmd/api/routes.go** file.

If you're following along, create this new file and add the following code:

```
$ touch cmd/api/routes.go
```

File: cmd/api/routes.go

```
package main

import (
    "net/http"

    "github.com/julienschmidt/httprouter"
)

func (app *application) routes() *httprouter.Router {
    // Initialize a new httprouter router instance.
    router := httprouter.New()

    // Register the relevant methods, URL patterns and handler functions for our
    // endpoints using the HandlerFunc() method. Note that http.MethodGet and
    // http.MethodPost are constants which equate to the strings "GET" and "POST"
    // respectively.
    router.HandlerFunc(http.MethodGet, "/v1/healthcheck", app.healthcheckHandler)
    router.HandlerFunc(http.MethodPost, "/v1/movies", app.createMovieHandler)
    router.HandlerFunc(http.MethodGet, "/v1/movies/:id", app.showMovieHandler)

    // Return the httprouter instance.
    return router
}
```

Hint: The `httprouter` package also provides a `router.Handler()` method which you can use when you want to register a regular `http.Handler` (rather than handler functions, like we are in the code above).

There are a couple of benefits to encapsulating our routing rules in this way. The first benefit is that it keeps our `main()` function clean and ensures all our routes are defined in one single place. The other big benefit, which we demonstrated in the first *Let's Go* book, is that we can now easily access the router in any test code by initializing an `application` instance and calling the `routes()` method on it.

The next thing that we need to do is update the `main()` function to remove the `http.ServeMux` declaration, and use the `httprouter` instance returned by `app.routes()` as our server handler instead. Like so:

File: cmd/api/main.go

```
package main

...

func main() {
    var cfg config

    flag.IntVar(&cfg.port, "port", 4000, "API server port")
    flag.StringVar(&cfg.env, "env", "development", "Environment (development|staging|production)")
    flag.Parse()

    logger := log.New(os.Stdout, "", log.Ldate|log.Ltime)

    app := &application{
        config: cfg,
        logger: logger,
    }

    // Use the httprouter instance returned by app.routes() as the server handler.
    srv := &http.Server{
        Addr:      fmt.Sprintf(":%d", cfg.port),
        Handler:   app.routes(),
        IdleTimeout: time.Minute,
        ReadTimeout: 10 * time.Second,
        WriteTimeout: 30 * time.Second,
    }

    logger.Printf("starting %s server on %s", cfg.env, srv.Addr)
    err := srv.ListenAndServe()
    logger.Fatal(err)
}
```

Adding the new handler functions

Now that the routing rules are set up, we need to make the `createMovieHandler` and `showMovieHandler` methods for the new endpoints. The `showMovieHandler` is particularly interesting here, because as part of this we want to extract the movie ID parameter from the URL and use it in our HTTP response.

Go ahead and create a new `cmd/api/movies.go` file to hold these two new handlers:

```
$ touch cmd/api/movies.go
```

And then add the following code:

File: cmd/api/movies.go

```
package main

import (
    "fmt"
    "net/http"
    "strconv"

    "github.com/julienschmidt/httprouter"
)

// Add a createMovieHandler for the "POST /v1/movies" endpoint. For now we simply
// return a plain-text placeholder response.
func (app *application) createMovieHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "create a new movie")
}

// Add a showMovieHandler for the "GET /v1/movies/:id" endpoint. For now, we retrieve
// the interpolated "id" parameter from the current URL and include it in a placeholder
// response.
func (app *application) showMovieHandler(w http.ResponseWriter, r *http.Request) {
    // When httprouter is parsing a request, any interpolated URL parameters will be
    // stored in the request context. We can use the ParamsFromContext() function to
    // retrieve a slice containing these parameter names and values.
    params := httprouter.ParamsFromContext(r.Context())

    // We can then use the ByName() method to get the value of the "id" parameter from
    // the slice. In our project all movies will have a unique positive integer ID, but
    // the value returned by ByName() is always a string. So we try to convert it to a
    // base 10 integer (with a bit size of 64). If the parameter couldn't be converted,
    // or is less than 1, we know the ID is invalid so we use the http.NotFound()
    // function to return a 404 Not Found response.
    id, err := strconv.ParseInt(params.ByName("id"), 10, 64)
    if err != nil || id < 1 {
        http.NotFound(w, r)
        return
    }

    // Otherwise, interpolate the movie ID in a placeholder response.
    fmt.Fprintf(w, "show the details of movie %d\n", id)
}
```

And with that, we're now ready to try this out!

Go ahead and restart the API application...

```
$ go run ./cmd/api
2021/04/06 08:57:25 starting development server on :4000
```

Then while the server is running, open a second terminal window and use `curl` to make some requests to the different endpoints. If everything is set up correctly, you will see some responses which look similar to this:

```
$ curl localhost:4000/v1/healthcheck
status: available
environment: development
version: 1.0.0

$ curl -X POST localhost:4000/v1/movies
create a new movie

$ curl localhost:4000/v1/movies/123
show the details of movie 123
```

Notice how, in the final example, the value of the movie `id` parameter `123` has been successfully retrieved from the URL and included in the response?

You might also want to try making some requests for a particular URL *using an unsupported HTTP method*. For example, let's try making a `POST` request to `/v1/healthcheck`:

```
$ curl -i -X POST localhost:4000/v1/healthcheck
HTTP/1.1 405 Method Not Allowed
Allow: GET, OPTIONS
Content-Type: text/plain; charset=utf-8
X-Content-Type-Options: nosniff
Date: Tue, 06 Apr 2021 06:59:04 GMT
Content-Length: 19

Method Not Allowed
```

That's looking really good. The `httprouter` package has automatically sent a `405 Method Not Allowed` response for us, including an `Allow` header which lists the HTTP methods that *are* supported for the endpoint.

Likewise, you can make an `OPTIONS` request to a specific URL and `httprouter` will send back a response with an `Allow` header detailing the supported HTTP methods. Like so:

```
$ curl -i -X OPTIONS localhost:4000/v1/healthcheck
HTTP/1.1 200 OK
Allow: GET, OPTIONS
Date: Tue, 06 Apr 2021 07:01:29 GMT
Content-Length: 0
```

Lastly, you might want to try making a request to the `GET /v1/movies/:id` endpoint with a negative number or a non-numerical `id` value in the URL. This should result in a `404 Not Found` response, similar to this:

```
$ curl -i localhost:4000/v1/movies/abc
HTTP/1.1 404 Not Found
Content-Type: text/plain; charset=utf-8
X-Content-Type-Options: nosniff
Date: Tue, 06 Apr 2021 07:02:01 GMT
Content-Length: 19

404 page not found
```

Creating a helper to read ID parameters

The code to extract an `id` parameter from a URL like `/v1/movies/:id` is something that we'll need repeatedly in our application, so let's abstract the logic for this into a small reusable helper method.

Go ahead and create a new `cmd/api/helpers.go` file:

```
$ touch cmd/api/helpers.go
```

And add a new `readIDParam()` method to the `application` struct, like so:

File: `cmd/api/helpers.go`

```
package main

import (
    "errors"
    "net/http"
    "strconv"

    "github.com/julienschmidt/httprouter"
)

// Retrieve the "id" URL parameter from the current request context, then convert it to
// an integer and return it. If the operation isn't successful, return 0 and an error.
func (app *application) readIDParam(r *http.Request) (int64, error) {
    params := httprouter.ParamsFromContext(r.Context())

    id, err := strconv.ParseInt(params.ByName("id"), 10, 64)
    if err != nil || id < 1 {
        return 0, errors.New("invalid id parameter")
    }

    return id, nil
}
```

Note: The `readIDParam()` method doesn't use any dependencies from our `application` struct so it *could* just be a regular function, rather than a method on `application`. But in general, I suggest setting up *all* your application-specific handlers and helpers so that they are methods on `application`. It helps maintain consistency in your code structure, and also future-proofs your code for when those handlers and helpers change later and they *do* need access to a dependency.

With this helper method in place, the code in our `showMovieHandler` can now be made a lot simpler:

```
File: cmd/api/movies.go

package main

import (
    "fmt"
    "net/http"
)

...

func (app *application) showMovieHandler(w http.ResponseWriter, r *http.Request) {
    id, err := app.readIDParam(r)
    if err != nil {
        http.NotFound(w, r)
        return
    }

    fmt.Fprintf(w, "show the details of movie %d\n", id)
}
```

Additional Information

Conflicting routes

It's important to be aware that `httprouter` doesn't allow *conflicting routes* which potentially match the same request. So, for example, you cannot register a route like `GET /foo/new` and another route with a parameter segment that conflicts with it — like `GET /foo/:id`.

If you're using a standard REST structure for your API endpoints — like we will be in this book — then this restriction is unlikely to cause you many problems.

In fact, it's arguably a positive thing. Because conflicting routes aren't allowed, there are no routing-priority rules that you need to worry about, and it reduces the risk of bugs and unintended behavior in your application.

But if you do need to support conflicting routes (for example, you might need to replicate the endpoints of an existing API exactly for backwards-compatibility), then I would recommend taking a look at [pat](#), [chi](#) or Gorilla [mux](#) instead. All of these are good routers which *do* permit conflicting routes.

Customizing httprouter behavior

The [httprouter](#) package provides a few configuration options that you can use to customize the behavior of your application further, including enabling *trailing slash redirects* and enabling *automatic URL path cleaning*.

More information about the available settings can be found [here](#).

Sending JSON Responses

In this section of the book, we're going to update our API handlers so that they return JSON responses instead of just plain text.

JSON (which is an acronym for *JavaScript Object Notation*) is a human-readable text format which can be used to represent structured data. As an example, in our project a movie could be represented with the following JSON:

```
{
  "id": 123,
  "title": "Casablanca",
  "runtime": 102,
  "genres": [
    "drama",
    "romance",
    "war"
  ],
  "version": 1
}
```

To give you a very quick overview of the JSON syntax...

- The parentheses `{ }` define a *JSON object*, which is made up of comma-separated key/value pairs.
- The keys must be strings, and the values can be either strings, numbers, booleans, other JSON objects, or an *array* of any of those things (enclosed by `[ ]`).
- Strings must be surrounded by double-quotes `"` (rather than single quotes `'`) and boolean values are either `true` or `false`.
- Outside of a string, whitespace characters in JSON *do not* carry any significance — it would be perfectly valid for us to represent the same movie using the following JSON instead:

```
{"id":123,"title":"Casablanca","runtime":102,"genres":["drama","romance","war"],"version":1}
```

If you've not used JSON at all before, then [this beginner's guide](#) provides a comprehensive introduction and I recommend reading it before following on.

In this section of the book you'll learn:

- How to send JSON responses from your REST API (including error responses).

- How to encode native Go objects into JSON using the [encoding/json](#) package.
- Different techniques for customizing how Go objects are encoded to JSON — first by using struct tags, and then by leveraging the [json.Marshaler](#) interface.
- How to create a reusable helper for sending JSON responses, which will ensure that all your API responses have a sensible and consistent structure.

Fixed-Format JSON

Let's begin by updating our `healthcheckHandler` to send a well-formed JSON response which looks like this:

```
{"status": "available", "environment": "development", "version": "1.0.0"}
```

At this stage, the thing I'd like to emphasize is that JSON is *just text*. Sure, it has certain control characters that give the text structure and meaning, but fundamentally, it is just text.

So that means you can write a JSON response from your Go handlers in the same way that you would write any other text response: using `w.Write()`, `io.WriteString()` or one of the `fmt.Fprint` functions.

In fact, the only special thing we need to do is set a `Content-Type: application/json` header on the response, so that the client knows it's receiving JSON and can interpret it accordingly.

Let's do exactly that.

Open up the `cmd/api/healthcheck.go` file and update the `healthcheckHandler` as follows:

File: cmd/api/healthcheck.go

```
package main

import (
    "fmt"
    "net/http"
)

func (app *application) healthcheckHandler(w http.ResponseWriter, r *http.Request) {
    // Create a fixed-format JSON response from a string. Notice how we're using a raw
    // string literal (enclosed with backticks) so that we can include double-quote
    // characters in the JSON without needing to escape them? We also use the %q verb to
    // wrap the interpolated values in double-quotes.
    js := `{"status": "available", "environment": %q, "version": %q}`
    js = fmt.Sprintf(js, app.config.env, version)

    // Set the "Content-Type: application/json" header on the response. If you forget to
    // this, Go will default to sending a "Content-Type: text/plain; charset=utf-8"
    // header instead.
    w.Header().Set("Content-Type", "application/json")

    // Write the JSON as the HTTP response body.
    w.Write([]byte(js))
}
```

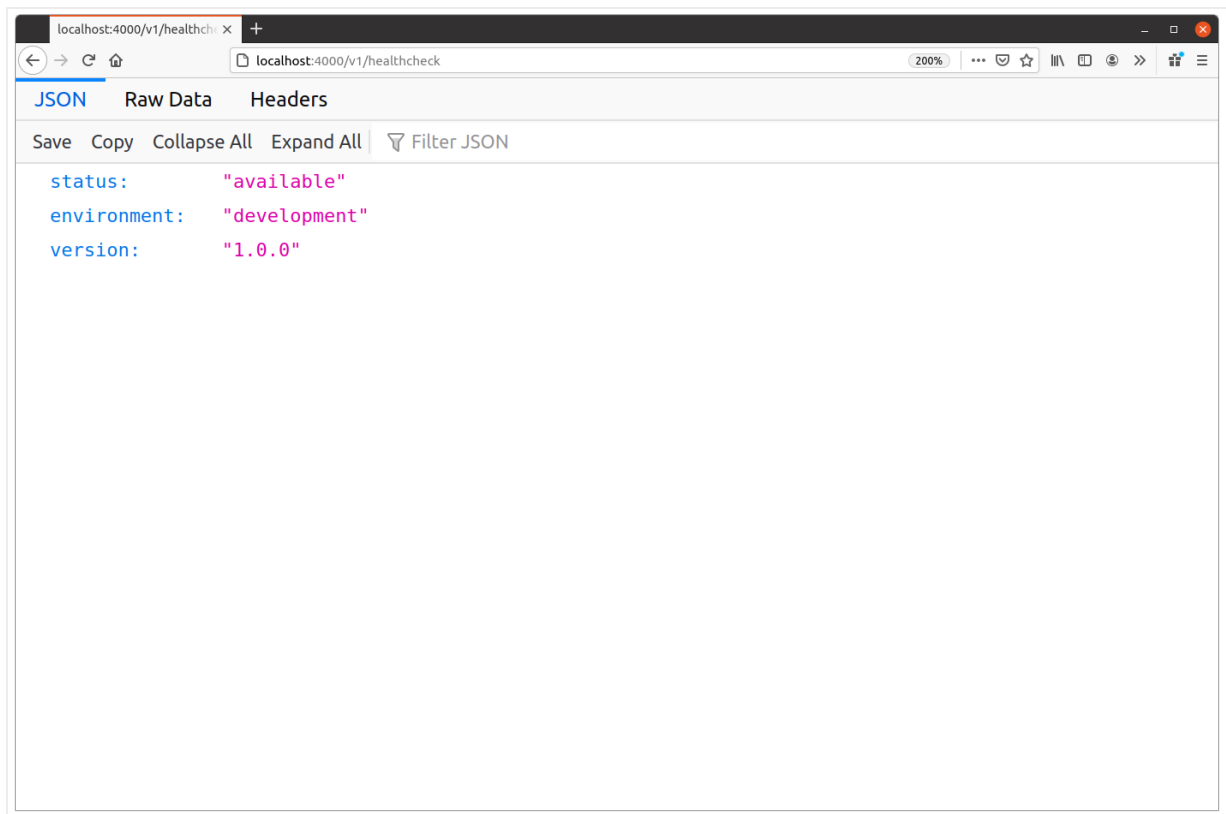
Once you've made those changes, restart the API, open a second terminal, and use `curl` to make a request to the `GET /v1/healthcheck` endpoint. You should now get back a response which looks like this:

```
$ curl -i localhost:4000/v1/healthcheck
HTTP/1.1 200 OK
Content-Type: application/json
Date: Tue, 06 Apr 2021 08:38:12 GMT
Content-Length: 73

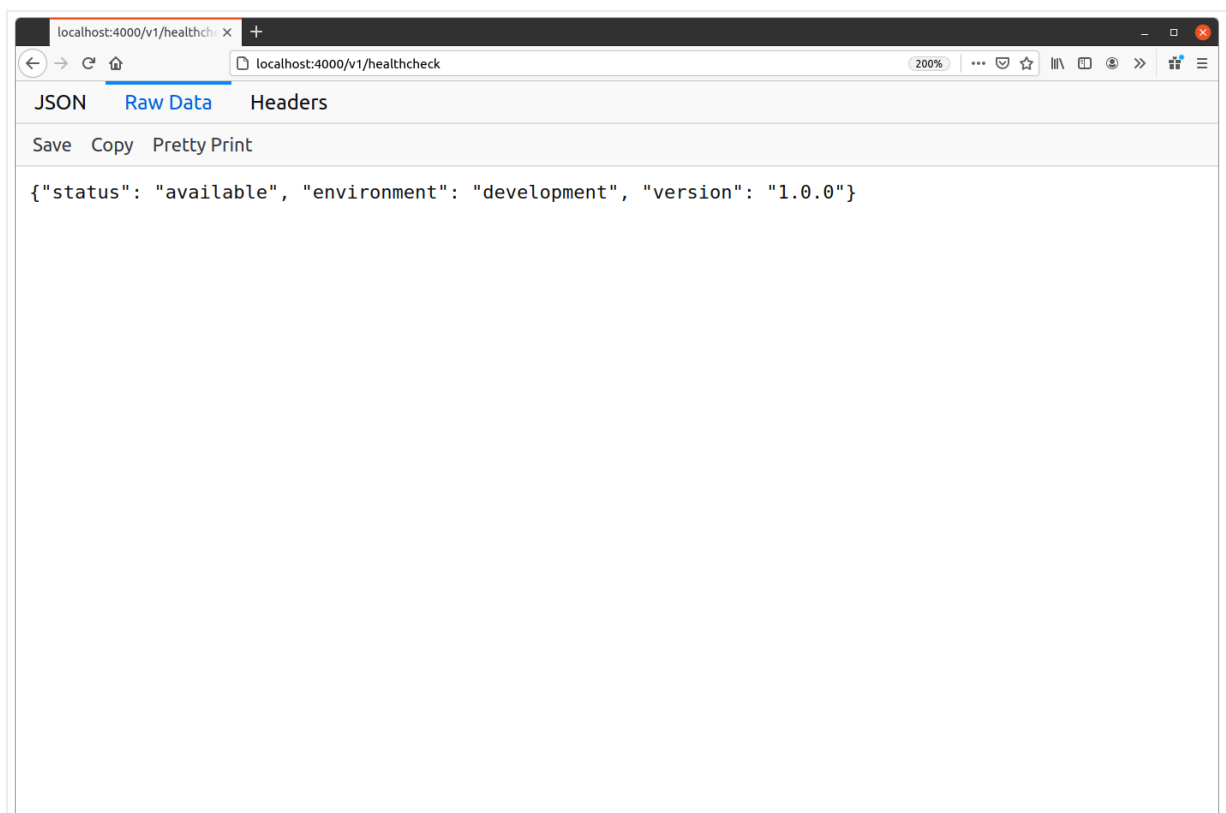
{"status": "available", "environment": "development", "version": "1.0.0"}
```

You might also like to try visiting `localhost:4000/v1/healthcheck` in your browser.

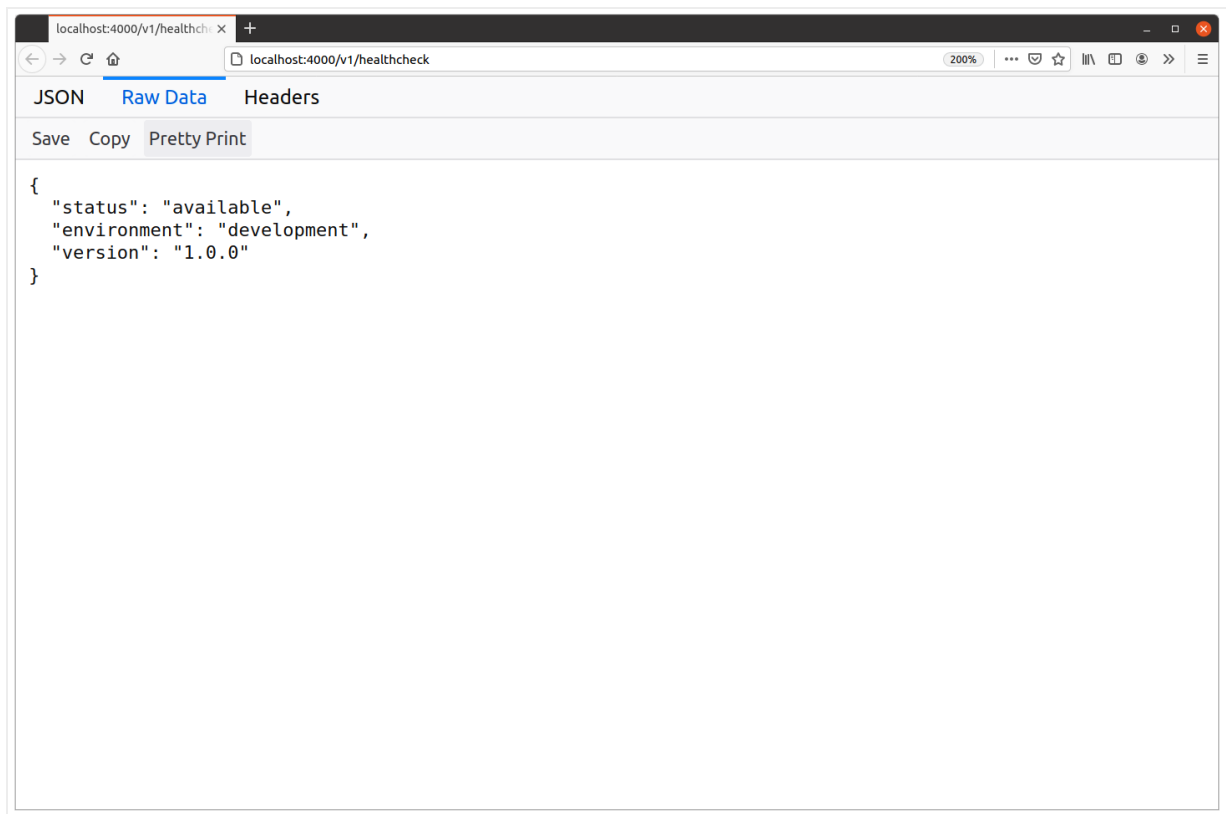
If you're running one of the newer versions of Firefox, you'll find that it knows the response contains JSON (due to the `Content-Type` header) and it will present the response using the inbuilt JSON viewer. Like so:



If you click on the **Raw Data** tab you should see the original unformatted JSON response:



Or you can even select the **Pretty Print** option to add whitespace, like so:



Of course, using a fixed-format string like we are in this chapter is a pretty simple and lo-fi approach to generating a JSON response. But it's worth remembering that it *is* a valid option. It can be useful for API endpoints that always return the same static JSON, or as a quick and easy way to generate small dynamic responses like we have here.

Additional Information

JSON charset

In your programming career you might have come across other JSON APIs which send responses with the header `Content-Type: application/json; charset=utf-8`.

Including a `charset` parameter like this isn't normally necessary. The [JSON RFC](#) states:

JSON text exchanged between systems that are not part of a closed ecosystem MUST be encoded using UTF-8.

The important word here is “must”. Because our API will be a public-facing application, it means that our JSON responses must always be UTF-8 encoded. And it also means that *it's safe for the client to assume that the responses it gets are always UTF-8 encoded*. Because of

this, including a `charset=utf-8` parameter is redundant.

The RFC [also explicitly notes](#) that the `application/json` media type does not have a `charset` parameter defined, which means that it is technically incorrect to include one too.

Basically, in our application a `charset` parameter serves no purpose, and it's safe (and also correct) to not include one in our `Content-Type: application/json` header.

JSON Encoding

Let's move on to something a bit more exciting and look at how to encode native Go objects (like maps, structs and slices) to JSON.

At a high-level, Go's `encoding/json` package provides two options for encoding things to JSON. You can either call the `json.Marshal()` function, or you can declare and use a `json.Encoder` type.

We'll explain how both approaches work in this chapter, but — for the purpose of sending JSON in a HTTP response — using `json.Marshal()` is generally the better choice. So let's start with that.

The way that `json.Marshal()` works is conceptually quite simple — you pass a native Go object to it as a parameter, and it returns a JSON representation of that object in a `[]byte` slice. The function signature looks like this:

```
func Marshal(v interface{}) ([]byte, error)
```

Note: The `v` parameter in the above method has the type `interface{}` (known as the `empty interface`). This effectively means that we're able to pass any Go type to `Marshal()` as the `v` parameter.

Let's jump in and update our `healthcheckHandler` so that it uses `json.Marshal()` to generate a JSON response directly from a Go map — instead of using a fixed-format string like we were before. Like so:

File: cmd/api/healthcheck.go

```
package main

import (
    "encoding/json" // New import
    "net/http"
)

func (app *application) healthcheckHandler(w http.ResponseWriter, r *http.Request) {
    // Create a map which holds the information that we want to send in the response.
    data := map[string]string{
        "status":    "available",
        "environment": app.config.env,
        "version":   version,
    }

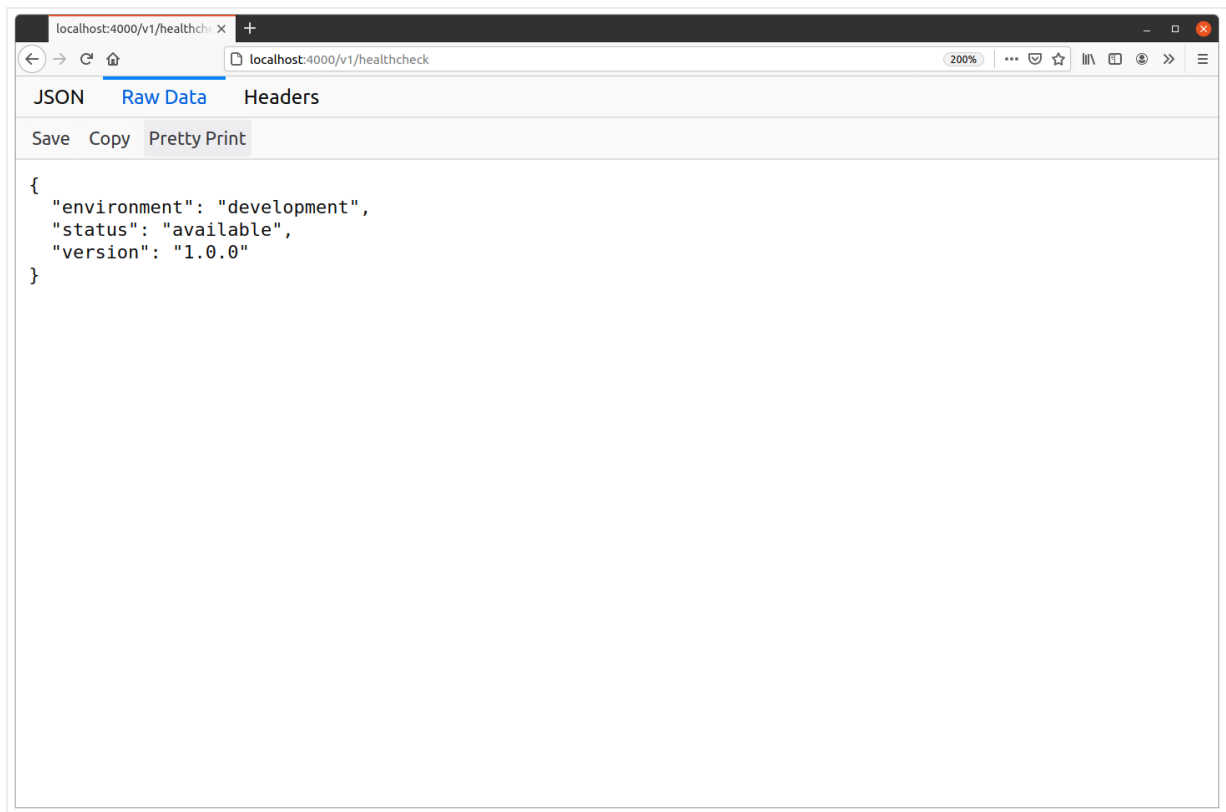
    // Pass the map to the json.Marshal() function. This returns a []byte slice
    // containing the encoded JSON. If there was an error, we log it and send the client
    // a generic error message.
    js, err := json.Marshal(data)
    if err != nil {
        app.logger.Println(err)
        http.Error(w, "The server encountered a problem and could not process your request", http.StatusInternalServerError)
        return
    }

    // Append a newline to the JSON. This is just a small nicety to make it easier to
    // view in terminal applications.
    js = append(js, '\n')

    // At this point we know that encoding the data worked without any problems, so we
    // can safely set any necessary HTTP headers for a successful response.
    w.Header().Set("Content-Type", "application/json")

    // Use w.Write() to send the []byte slice containing the JSON as the response body.
    w.Write(js)
}
```

If you restart the API and visit localhost:4000/v1/healthcheck in your browser, you should now get a response similar to this:



That's looking good — we can see that the map has automatically been encoded to a JSON object for us, with the key/value pairs in the map appearing as *alphabetically sorted* key/value pairs in the JSON object.

Creating a writeJSON helper method

As our API grows we're going to be sending a lot of JSON responses, so it makes sense to move some of this logic into a reusable `writeJSON()` helper method.

As well as creating and sending the JSON, we want to design this helper so that we can include *arbitrary headers* in successful responses later, such as a `Location` header after creating a new movie in our system.

If you're coding-along, open the `cmd/api/helpers.go` file again and create the following `writeJSON()` method:

File: cmd/api/helpers.go

```
package main

import (
    "encoding/json" // New import
    "errors"
    "net/http"
    "strconv"

    "github.com/julienschmidt/httprouter"
)

...

// Define a writeJSON() helper for sending responses. This takes the destination
// http.ResponseWriter, the HTTP status code to send, the data to encode to JSON, and a
// header map containing any additional HTTP headers we want to include in the response.
func (app *application) writeJSON(w http.ResponseWriter, status int, data interface{}, headers http.Header) error {
    // Encode the data to JSON, returning the error if there was one.
    js, err := json.Marshal(data)
    if err != nil {
        return err
    }

    // Append a newline to make it easier to view in terminal applications.
    js = append(js, '\n')

    // At this point, we know that we won't encounter any more errors before writing the
    // response, so it's safe to add any headers that we want to include. We loop
    // through the header map and add each header to the http.ResponseWriter header map.
    // Note that it's OK if the provided header map is nil. Go doesn't throw an error
    // if you try to range over (or generally, read from) a nil map.
    for key, value := range headers {
        w.Header()[key] = value
    }

    // Add the "Content-Type: application/json" header, then write the status code and
    // JSON response.
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(status)
    w.Write(js)

    return nil
}
```

Now that the `writeJSON()` helper is in place, we can significantly simplify the code in `healthcheckHandler`, like so

File: cmd/api/healthcheck.go

```
package main

import (
    "net/http"
)

func (app *application) healthcheckHandler(w http.ResponseWriter, r *http.Request) {
    data := map[string]string{
        "status":      "available",
        "environment": app.config.env,
        "version":     version,
    }

    err := app.writeJSON(w, http.StatusOK, data, nil)
    if err != nil {
        app.logger.Println(err)
        http.Error(w, "The server encountered a problem and could not process your request", http.StatusInternalServerError)
    }
}
```

If you run the application again now, everything will compile correctly and a request to the `GET /v1/healthcheck` endpoint should result in the same HTTP response as before.

Additional Information

How different Go types are encoded

In this chapter we've been encoding a `map[string]string` type to JSON, which resulted in a JSON object with JSON strings as the values in the key/value pairs. But Go supports encoding many other native types too.

The following table summarizes how different Go types are mapped to JSON data types during encoding:

Go type	⇒	JSON type
<code>bool</code>	⇒	JSON boolean
<code>string</code>	⇒	JSON string
<code>int*</code> , <code>uint*</code> , <code>float*</code> , <code>rune</code>	⇒	JSON number
<code>array</code> , <code>slice</code>	⇒	JSON array
<code>struct</code> , <code>map</code>	⇒	JSON object
<code>nil</code> pointers, interface values, slices, maps, etc.	⇒	JSON null
<code>chan</code> , <code>func</code> , <code>complex*</code>	⇒	Not supported
<code>time.Time</code>	⇒	RFC3339-format JSON string
<code>[]byte</code>	⇒	Base64-encoded JSON string

The last two of these are special cases which deserve a bit more explanation:

- Go `time.Time` values (which are actually a `struct` behind the scenes) will be encoded as a JSON string in RFC 3339 format like `"2020-11-08T06:27:59+01:00"`, rather than as a JSON object.
- A `[]byte` slice will be encoded as a base64-encoded JSON string, rather than as a JSON array. So, for example, a byte slice of `[]byte{'h','e','l','l','o'}` would appear as `"aGVsbG8="` in the JSON output. The base64 encoding uses padding and the [standard character set](#).

A few other important things to mention:

- Encoding of nested objects is supported. So, for example, if you have a *slice of structs* in Go that will encode to an *array of objects* in JSON.
- Channels, functions and `complex` number types cannot be encoded. If you try to do so, you'll get a `json.UnsupportedTypeError` error at runtime.
- Any pointer values will encode as *the value pointed to*. Likewise, `interface{}` values will encode as the value contained in the interface.

Using `json.Encoder`

At the start of this chapter I mentioned that it's also possible to use Go's `json.Encoder` type to perform the encoding. This allows you to encode an object to JSON *and write that JSON*

to an output stream in a single step.

For example, you could use it in a handler like this:

```
func (app *application) exampleHandler(w http.ResponseWriter, r *http.Request) {
    data := map[string]string{
        "hello": "world",
    }

    // Set the "Content-Type: application/json" header on the response.
    w.Header().Set("Content-Type", "application/json")

    // Use the json.NewEncoder() function to initialize a json.Encoder instance that
    // writes to the http.ResponseWriter. Then we call its Encode() method, passing in
    // the data that we want to encode to JSON (which in this case is the map above). If
    // the data can be successfully encoded to JSON, it will then be written to our
    // http.ResponseWriter.
    err := json.NewEncoder(w).Encode(data)
    if err != nil {
        app.logger.Println(err)
        http.Error(w, "The server encountered a problem and could not process your request", http.StatusInternalServerError)
    }
}
```

This pattern works, and it's very neat and elegant, but if you consider it carefully you might notice a slight problem...

When we call `json.NewEncoder(w).Encode(data)` the JSON is created and written to the `http.ResponseWriter` in a single step, which means there's no opportunity to set HTTP response headers conditionally based on whether the `Encode()` method returns an error or not.

Imagine, for example, that you want to set a `Cache-Control` header on a successful response, *but not set a `Cache-Control` header if the JSON encoding fails and you have to return an error response.*

Implementing that cleanly while using the `json.Encoder` pattern is quite difficult.

You *could* set the `Cache-Control` header and then *delete it from the header map again* in the event of an error — but that's pretty hacky.

Another option is to write the JSON to an interim `bytes.Buffer` instead of directly to the `http.ResponseWriter`. You can then check for any errors, before setting the `Cache-Control` header and copying the JSON from the `bytes.Buffer` to `http.ResponseWriter`. But once you start doing that, it's simpler and cleaner (as well as slightly faster) to use the alternative `json.Marshal()` approach instead.

Performance of `json.Encoder` and `json.Marshal`

Talking of speed, you might be wondering if there's any performance difference between using `json.Encoder` and `json.Marshal()`. The short answer to that is yes... but the difference is small and in most cases you should not worry about it.

The following benchmarks demonstrate the performance of the two approaches using the code in [this gist](#) (note that each benchmark test is repeated three times):

```
$ go test -run=^$ -bench=. -benchmem -count=3 -benchtime=5s
goos: linux
goarch: amd64
BenchmarkEncoder-8      3477318      1692 ns/op      1046 B/op      15 allocs/op
BenchmarkEncoder-8      3435145      1704 ns/op      1048 B/op      15 allocs/op
BenchmarkEncoder-8      3631305      1595 ns/op      1039 B/op      15 allocs/op
BenchmarkMarshal-8      3624570      1616 ns/op      1119 B/op      16 allocs/op
BenchmarkMarshal-8      3549090      1626 ns/op      1123 B/op      16 allocs/op
BenchmarkMarshal-8      3548070      1638 ns/op      1123 B/op      16 allocs/op
```

In these results we can see that `json.Marshal()` requires ever so slightly more memory (B/op) than `json.Encoder`, and also makes one extra heap memory allocation (allocs/op).

There's no obvious observable difference in the average runtime (ns/op) between the two approaches. Perhaps with a larger benchmark sample or a larger data set a difference might become clear, but it's likely to be in the order of *microseconds*, rather than anything larger.

Additional JSON encoding nuances

Encoding things to JSON in Go is mostly quite intuitive. But there are a handful of behaviors which might either catch you out or surprise you when you first encounter them.

We've already mentioned a couple of these in this chapter (in particular — *map entries being sorted alphabetically* and *byte slices being base-64 encoded*), but I've included a full list in [this appendix](#).

Encoding Structs

In this chapter we're going to head back to the `showMovieHandler` method that we made earlier and update it to return a JSON response which represents a single movie in our system. Similar to this:

```
{
  "id": 123,
  "title": "Casablanca",
  "runtime": 102,
  "genres": [
    "drama",
    "romance",
    "war"
  ],
  "version": 1
}
```

Instead of encoding a map to create this JSON object (like we did in the previous chapter), this time we're going to encode a custom `Movie` struct.

So, first things first, we need to begin by defining a custom `Movie` struct. We'll do this inside a new `internal/data` package, which later will grow to encapsulate *all* the custom data types for our project along with the logic for interacting with our database.

If you're following along, go ahead and create a new `internal/data` directory containing a `movies.go` file:

```
$ mkdir internal/data
$ touch internal/data/movies.go
```

And in this new file let's define the custom `Movie` struct, like so:

File: internal/data/movies.go

```
package data

import (
    "time"
)

type Movie struct {
    ID          int64    // Unique integer ID for the movie
    CreatedAt   time.Time // Timestamp for when the movie is added to our database
    Title       string   // Movie title
    Year        int32    // Movie release year
    Runtime     int32    // Movie runtime (in minutes)
    Genres      []string // Slice of genres for the movie (romance, comedy, etc.)
    Version     int32    // The version number starts at 1 and will be incremented each
                                // time the movie information is updated
}
```

Important: It's crucial to point out here that all the fields in our `Movie` struct are exported (i.e. start with a capital letter), which is necessary for them to be visible to Go's `encoding/json` package. Any fields which *aren't* exported won't be included when encoding a struct to JSON.

Now that's done, let's update our `showMovieHandler` to initialize an instance of the `Movie` struct containing some dummy data, and then send it as a JSON response using our `writeJSON()` helper.

It's quite simple in practice:

File: cmd/api/movies.go

```
package main

import (
    "fmt"
    "net/http"
    "time" // New import

    "greenlight.alexedwards.net/internal/data" // New import
)

...

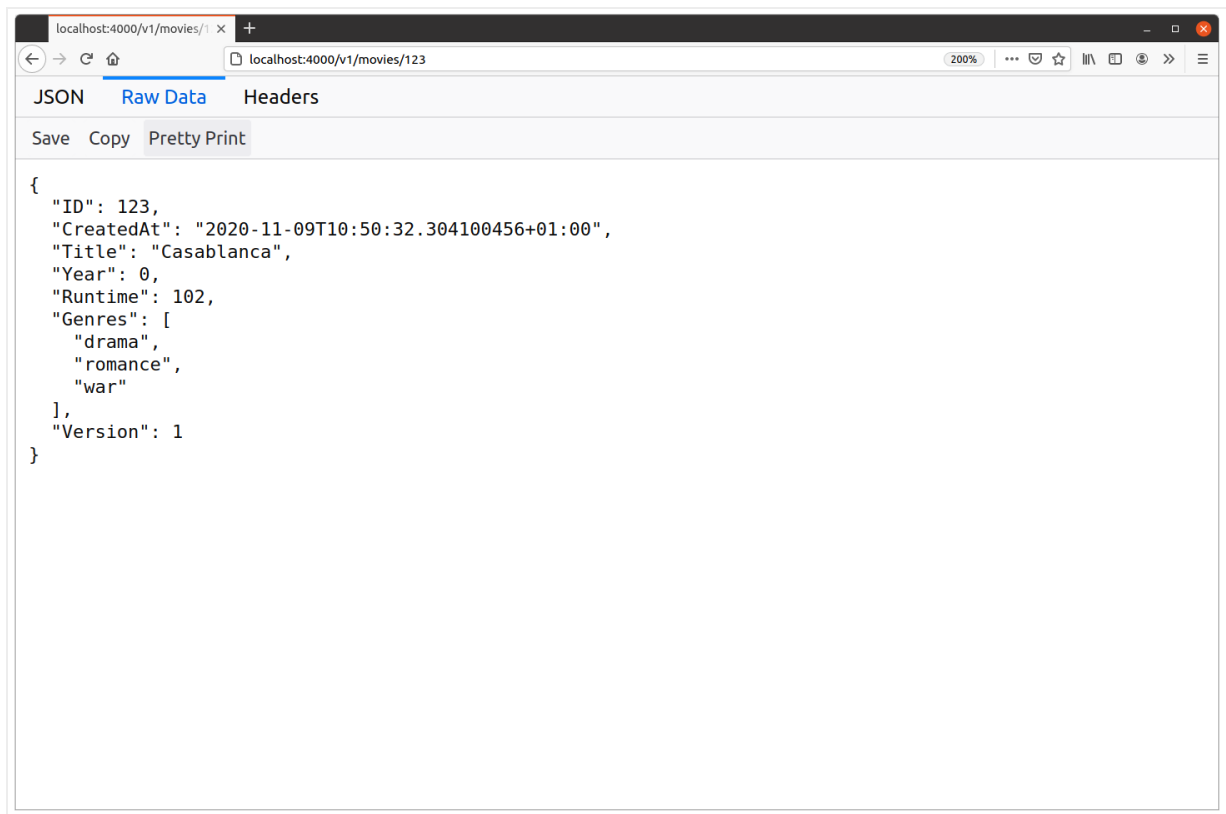
func (app *application) showMovieHandler(w http.ResponseWriter, r *http.Request) {
    id, err := app.readIDParam(r)
    if err != nil {
        http.NotFound(w, r)
        return
    }

    // Create a new instance of the Movie struct, containing the ID we extracted from
    // the URL and some dummy data. Also notice that we deliberately haven't set a
    // value for the Year field.
    movie := data.Movie{
        ID:      id,
        CreatedAt: time.Now(),
        Title:    "Casablanca",
        Runtime:  102,
        Genres:   []string{"drama", "romance", "war"},
        Version:  1,
    }

    // Encode the struct to JSON and send it as the HTTP response.
    err = app.writeJSON(w, http.StatusOK, movie, nil)
    if err != nil {
        app.logger.Println(err)
        http.Error(w, "The server encountered a problem and could not process your request", http.StatusInternalServerError)
    }
}
```

OK, let's give this a try!

Restart the API and then visit localhost:4000/v1/movies/123 in your browser. You should see a JSON response which looks similar to this:



There are a few interesting things in this response to point out:

- Our `Movie` struct has been encoded into a single JSON object, with the field names and values as the key/value pairs.
- By default, the keys in the JSON object are equal to the field names in the struct (`ID`, `CreatedAt`, `Title` and so on). We'll talk about how to customize these shortly.
- If a struct field doesn't have an explicit value set, then the JSON-encoding of the *zero value* for the field will appear in the output. We can see an example of this in the response above — we didn't set a value for the `Year` field in our Go code, but it still appears in the JSON output with the value `0`.

Changing keys in the JSON object

One of the nice things about encoding structs in Go is that you can customize the JSON by annotating the fields with [struct tags](#).

Probably the most common use of struct tags is to change the key names that appear in the JSON object. This can be useful when your struct field names aren't appropriate for public-facing responses, or you want to use an alternative [casing style](#) in your JSON output.

To illustrate how to do this, let's annotate our `Movies` struct with struct tags so that it uses

`snake_case` for the keys instead. Like so:

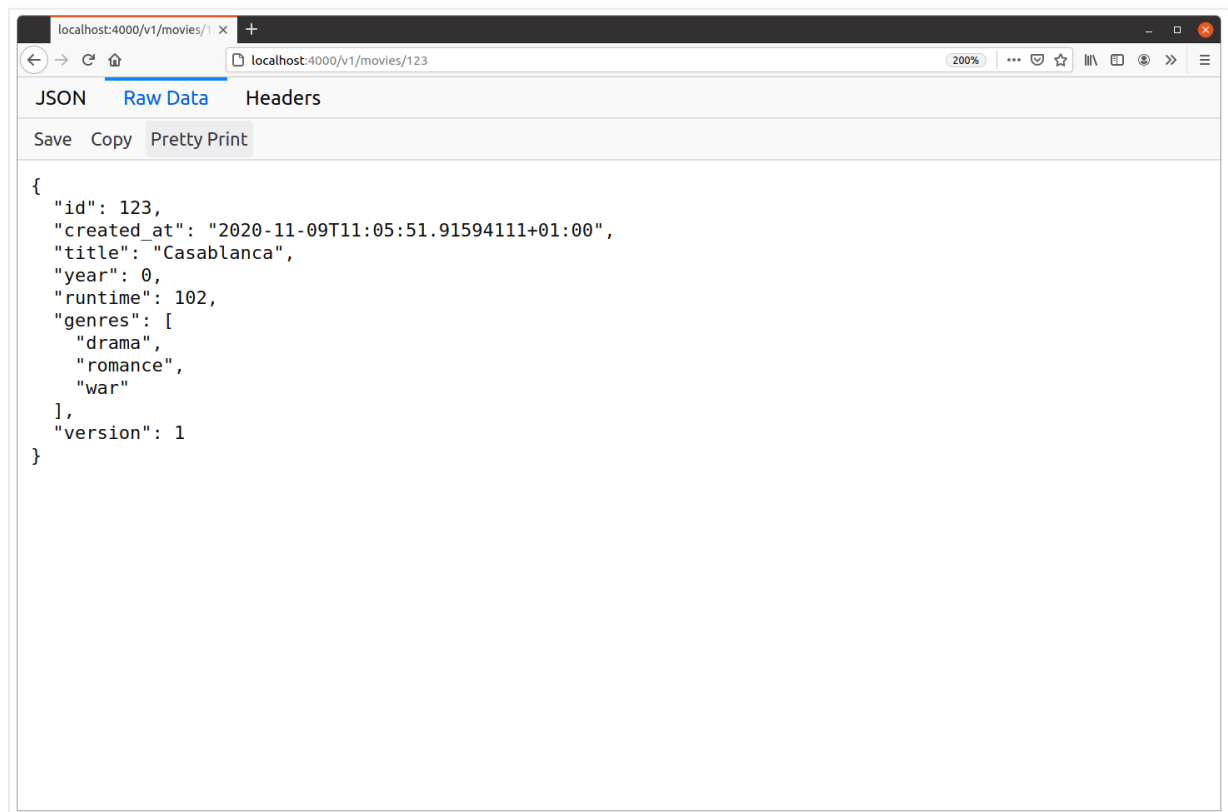
File: internal/data/movies.go

```
package data

...

// Annotate the Movie struct with struct tags to control how the keys appear in the
// JSON-encoded output.
type Movie struct {
    ID          int64    `json:"id"`
    CreatedAt   time.Time `json:"created_at"`
    Title       string    `json:"title"`
    Year        int32     `json:"year"`
    Runtime     int32     `json:"runtime"`
    Genres      []string  `json:"genres"`
    Version     int32     `json:"version"`
}
```

And if you restart the server and visit localhost:4000/v1/movies/123 again, you should now see a response with `snake_case` keys similar to this:



Hiding struct fields in the JSON object

It's also possible to control the visibility of individual struct fields in the JSON by using the

`omitempty` and `-` struct tag directives.

The `-` (hyphen) directive can be used when you *never* want a particular struct field to appear in the JSON output. This is useful for fields that contain internal system information that isn't relevant to your users, or sensitive information that you don't want to expose (like the hash of a password).

In contrast the `omitempty` directive hides a field in the JSON output *if and only if* the struct field value is empty, where empty is defined as being:

- Equal to `false`, `0`, or `""`
- An empty `array`, `slice` or `map`
- A `nil` pointer or a `nil` interface value

To demonstrate how to use these directives, let's make a couple more changes to our `Movie` struct. The `CreatedAt` field isn't relevant to our end users, so let's always hide this in the output using the `-` directive. And we'll also use the `omitempty` directive to hide the `Year`, `Runtime` and `Genres` fields in the output *if and only if* they are empty.

Go ahead and update the struct tags like so:

File: internal/data/movies.go

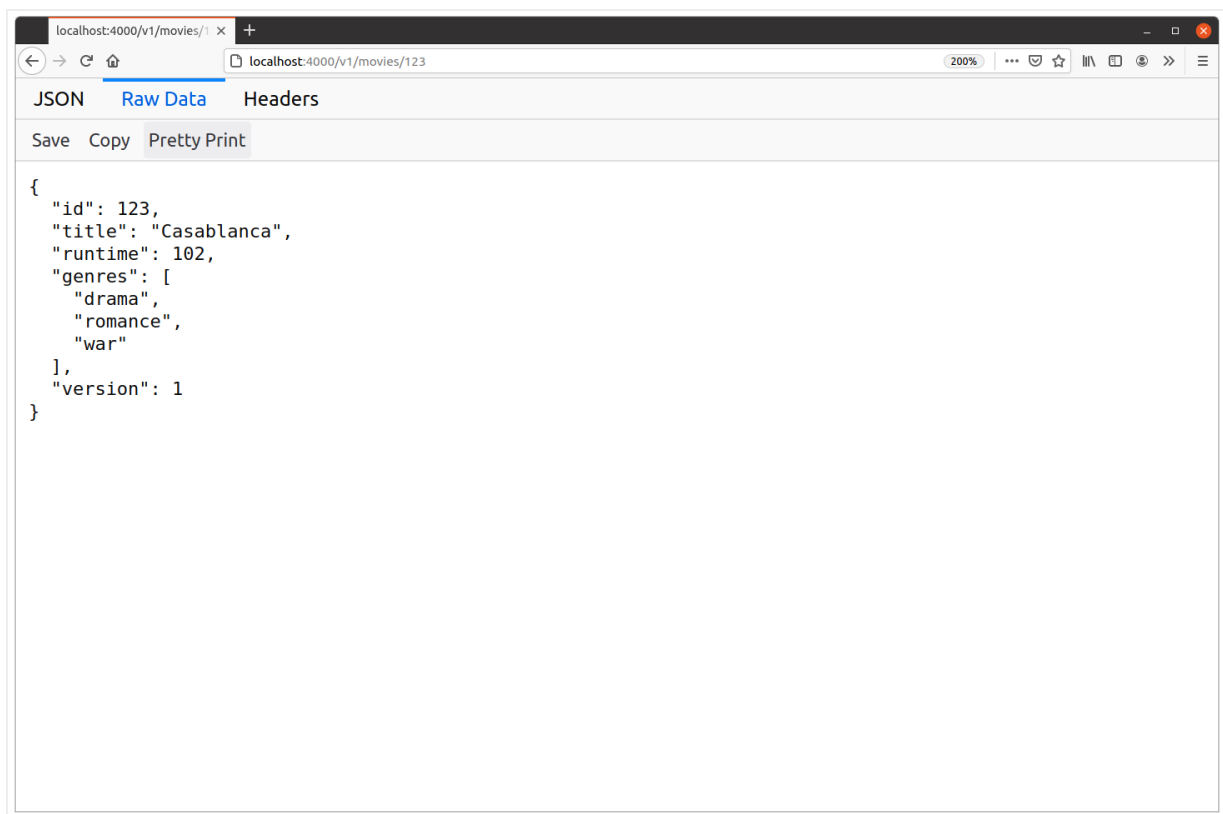
```
package data

...

type Movie struct {
    ID          int64    `json:"id"`
    CreatedAt   time.Time `json:"- "` // Use the - directive
    Title       string   `json:"title"`
    Year        int32    `json:"year,omitempty"` // Add the oitempty directive
    Runtime     int32    `json:"runtime,omitempty"` // Add the oitempty directive
    Genres      []string `json:"genres,omitempty"` // Add the oitempty directive
    Version     int32    `json:"version"`
}
```

Hint: If you want to use `omitempty` and *not* change the key name then you can leave it blank in the struct tag — like this: `json:",omitempty"`. Notice that the leading comma is still required.

Now when you restart the application and refresh your web browser, you should see a response which looks exactly like this:



We can see here that the `CreatedAt` struct field no longer appears in the JSON at all, and the `Year` field (which had the value `0`) doesn't appear either thanks to the `omitempty` directive. The other fields that we used `omitempty` on (`Runtime` and `Genres`) are unaffected.

Note: You can also prevent a struct field from appearing in the JSON output by simply making it unexported. But using the `json:"-"` struct tag is generally a better choice: it's an explicit indication to both Go and any future readers of your code that you don't want the field included in the JSON, and it helps prevent problems if someone changes the field to be exported in the future without realizing the consequences.

Old versions of `go vet` used to raise an error if you tried to use a struct tag on an unexported field, but this has [been fixed](#) as of Go 1.16.

Additional Information

The string struct tag directive

A final, less-frequently-used, struct tag directive is `string`. You can use this on individual

struct fields to force the data to be represented as a string in the JSON output.

For example, if we wanted the value of our `Runtime` field to be represented as a JSON string (instead of a number) we could use the `string` directive like this:

```
type Movie struct {
    ID          int64    `json:"id"`
    CreatedAt   time.Time `json:"-"`
    Title       string   `json:"title"`
    Year        int32    `json:"year,omitempty"`
    Runtime     int32    `json:"runtime,omitempty,string" // Add the string directive
    Genres      []string `json:"genres,omitempty"`
    Version     int32    `json:"version"`
}
```

And the resulting JSON output would look like this:

```
{
  "id": 123,
  "title": "Casablanca",
  "runtime": "102",      ← This is now a string
  "genres": [
    "drama",
    "romance",
    "war"
  ],
  "version": 1
}
```

Note that the `string` directive will only work on struct fields which have `int*`, `uint*`, `float*` or `bool` types. For any other type of struct field it will have no effect.

Formatting and Enveloping Responses

So far in this book we've generally been making requests to our API using Firefox, which makes the JSON responses easy-to-read thanks to the 'pretty printing' provided by the built-in JSON viewer.

But if you try making some requests using curl, you'll see that the actual JSON response data is all just on one line with no whitespace.

```
$ curl localhost:4000/v1/healthcheck
{"environment":"development","status":"available","version":"1.0.0"}

$ curl localhost:4000/v1/movies/123
{"id":123,"title":"Casablanca","runtime":102,"genres":["drama","romance","war"],"version":1}
```

We can make these easier to read in terminals by using the `json.MarshalIndent()` function to encode our response data, instead of the regular `json.Marshal()` function. This automatically adds whitespace to the JSON output, putting each element on a separate line and prefixing each line with optional *prefix* and *indent* characters.

Let's update our `writeJSON()` helper to use this instead:

```
File: cmd/api/helpers.go

package main

...

func (app *application) writeJSON(w http.ResponseWriter, status int, data interface{}, headers http.Header) error {
    // Use the json.MarshalIndent() function so that whitespace is added to the encoded
    // JSON. Here we use no line prefix ("" ) and tab indents ("\t") for each element.
    js, err := json.MarshalIndent(data, "", "\t")
    if err != nil {
        return err
    }

    js = append(js, '\n')

    for key, value := range headers {
        w.Header()[key] = value
    }

    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(status)
    w.Write(js)

    return nil
}
```

If you restart the API and try making the same requests from your terminal again, you will now receive some nicely-whitespaced JSON responses similar to these:

```
$ curl -i localhost:4000/v1/healthcheck
{
  "environment": "development",
  "status": "available",
  "version": "1.0.0"
}

$ curl localhost:4000/v1/movies/123
{
  "id": 123,
  "title": "Casablanca",
  "runtime": 102,
  "genres": [
    "drama",
    "romance",
    "war"
  ],
  "version": 1
}
```

Relative performance

While using `json.MarshalIndent()` is positive from a readability and user-experience point of view, it unfortunately doesn't come for free. As well as the fact that the responses are now slightly larger in terms of total bytes, the extra work that Go does to add the whitespace has a notable impact on performance.

The following benchmarks help to demonstrate the relative performance of `json.Marshal()` and `json.MarshalIndent()` using the code in [this gist](#).

```
$ go test -run=^$ -bench=. -benchmem -count=3 -benchtime=5s
goos: linux
goarch: amd64
BenchmarkMarshalIndent-8      2177511      2695 ns/op    1472 B/op    18 allocs/op
BenchmarkMarshalIndent-8      2170448      2677 ns/op    1473 B/op    18 allocs/op
BenchmarkMarshalIndent-8      2150780      2712 ns/op    1476 B/op    18 allocs/op
BenchmarkMarshal-8            3289424      1681 ns/op    1135 B/op    16 allocs/op
BenchmarkMarshal-8            3532242      1641 ns/op    1123 B/op    16 allocs/op
BenchmarkMarshal-8            3619472      1637 ns/op    1119 B/op    16 allocs/op
```

In these benchmarks we can see that `json.MarshalIndent()` takes 65% longer to run and uses around 30% more memory than `json.Marshal()`, as well as making two more heap allocations. Those figures will change depending on what you're encoding, but in my experience they're fairly indicative of the performance impact.

For most applications this performance difference simply isn't something that you need to

worry about. In real terms we're talking about a few thousandths of a *millisecond* — and the improved readability of responses is probably worth this trade-off. But if your API is operating in a very resource-constrained environment, or needs to manage extremely high levels of traffic, then this is worth being aware of and you may prefer to stick with using `json.Marshal()` instead.

Note: Behind the scenes `json.MarshalIndent()` works by calling `json.Marshal()` as normal, then running the JSON through the standalone `json.Indent()` function to add the whitespace. There's also a reverse function available, `json.Compact()`, which you can use to remove whitespace from JSON.

Enveloping responses

Next let's work on updating our responses so that the JSON data is always *enveloped* in a parent JSON object. Similar to this:

```
{
  "movie": {
    "id": 123,
    "title": "Casablanca",
    "runtime": 102,
    "genres": [
      "drama",
      "romance",
      "war"
    ],
    "version": 1
  }
}
```

Notice how the movie data is nested under the key `"movie"` here, rather than being the top-level JSON object itself?

Enveloping response data like this isn't strictly necessary, and whether you choose to do so is partly a matter of style and taste. But there are a few tangible benefits:

1. Including a key name (like `"movie"`) at the top-level of the JSON helps make the response more self-documenting. For any humans who see the response out of context, it is a bit easier to understand what the data relates to.
2. It reduces the risk of errors on the client side, because it's harder to accidentally process one response thinking that it is something different. To get at the data, a client must explicitly reference it via the `"movie"` key.

3. If we always envelope the data returned by our API, then we mitigate a [security vulnerability](#) in older browsers which can arise if you return a JSON array as a response.

There are a couple of techniques that we could use to envelope our API responses, but we're going to keep things simple and do it by creating a custom `envelope` map with the underlying type `map[string]interface{}`.

I'll demonstrate.

Let's start by updating the `cmd/api/helpers.go` file as follows:

File: `cmd/api/helpers.go`

```
package main

...

// Define an envelope type.
type envelope map[string]interface{}

// Change the data parameter to have the type envelope instead of interface{}.
func (app *application) writeJSON(w http.ResponseWriter, status int, data envelope, headers http.Header) error {
    js, err := json.MarshalIndent(data, "", "\t")
    if err != nil {
        return err
    }

    js = append(js, '\n')

    for key, value := range headers {
        w.Header()[key] = value
    }

    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(status)
    w.Write(js)

    return nil
}
```

Then we need to update our `showMovieHandler` to create an instance of the `envelope` map containing the movie data, and pass this onwards to our `writeJSON()` helper instead of passing the movie data directly.

Like so:

File: cmd/api/movies.go

```
package main

...

func (app *application) showMovieHandler(w http.ResponseWriter, r *http.Request) {
    id, err := app.readIDParam(r)
    if err != nil {
        http.NotFound(w, r)
        return
    }

    movie := data.Movie{
        ID:      id,
        CreatedAt: time.Now(),
        Title:   "Casablanca",
        Runtime: 102,
        Genres:  []string{"drama", "romance", "war"},
        Version: 1,
    }

    // Create an envelope{"movie": movie} instance and pass it to writeJSON(), instead
    // of passing the plain movie struct.
    err = app.writeJSON(w, http.StatusOK, envelope{"movie": movie}, nil)
    if err != nil {
        app.logger.Println(err)
        http.Error(w, "The server encountered a problem and could not process your request", http.StatusInternalServerError)
    }
}
```

We also need to update the code in our `healthcheckHandler` so that it passes an `envelope` type to the `writeJSON()` helper too:

File: cmd/api/healthcheck.go

```
package main

...

func (app *application) healthcheckHandler(w http.ResponseWriter, r *http.Request) {
    // Declare an envelope map containing the data for the response. Notice that the way
    // we've constructed this means the environment and version data will now be nested
    // under a system_info key in the JSON response.
    env := envelope{
        "status": "available",
        "system_info": map[string]string{
            "environment": app.config.env,
            "version":     version,
        },
    }

    err := app.writeJSON(w, http.StatusOK, env, nil)
    if err != nil {
        app.logger.Println(err)
        http.Error(w, "The server encountered a problem and could not process your request", http.StatusInternalServerError)
    }
}
```

Alright, let's try these changes out. Go ahead and restart the server, then use `curl` to make some requests to the API endpoints again. You should now get responses formatted like the ones below.

```
$ curl localhost:4000/v1/movies/123
{
  "movie": {
    "id": 123,
    "title": "Casablanca",
    "runtime": 102,
    "genres": [
      "drama",
      "romance",
      "war"
    ],
    "version": 1
  }
}

$ curl localhost:4000/v1/healthcheck
{
  "status": "available",
  "system_info": {
    "environment": "development",
    "version": "1.0.0"
  }
}
```

Additional Information

Response structure

It's important to emphasize that there's no single right or wrong way to structure your JSON responses. There are some popular formats like [JSON:API](#) and [jsend](#) that you might like to follow or use for inspiration, but it's certainly not necessary and most APIs *don't* follow these formats.

But whatever you do, it *is* valuable to think about formatting upfront and to maintain a clear and consistent response structure across your different API endpoints — especially if they are being made available for public use.

Advanced JSON Customization

By using struct tags, adding whitespace and enveloping the data, we've been able to add quite a lot of customization to our JSON responses already. But what happens when these things aren't enough, and you need the freedom to customize your JSON even more?

To answer this question, we first need to talk some theory about how Go handles JSON encoding behind the scenes. The key thing to understand is this:

When Go is encoding a particular type to JSON, it looks to see if the type has a `MarshalJSON()` method implemented on it. If it has, then Go will call this method to determine how to encode it.

That language is a bit fuzzy, so let's be more exact.

Strictly speaking, when Go is encoding a particular type to JSON it looks to see if the type satisfies the `json.Marshaler` interface, which looks like this:

```
type Marshaler interface {  
    MarshalJSON() ([]byte, error)  
}
```

If the type does satisfy the interface, then Go will call its `MarshalJSON()` method and use the `[]byte` slice that it returns as the encoded JSON value.

If the type doesn't have a `MarshalJSON()` method, then Go will fall back to trying to encode it to JSON based on its own internal set of rules.

So, if we want to customize how something is encoded, all we need to do is implement a `MarshalJSON()` method on it which returns a *custom JSON representation of itself* in a `[]byte` slice.

Hint: You can see this in action if you look at the source code for Go's `time.Time` type. Behind the scenes `time.Time` is *actually a struct*, but it has a `MarshalJSON()` method which outputs a RFC 3339 format representation of itself. This is what gets called whenever a `time.Time` value is encoded to JSON.

Customizing the Runtime field

To help illustrate this, let's look at a concrete example in our application.

When our `Movie` struct is encoded to JSON, the `Runtime` field (which is an `int32` type) is currently formatted as a JSON number. Let's change this so that it's encoded as a string with the format "`<runtime> mins`" instead. Like so:

```
{
  "id": 123,
  "title": "Casablanca",
  "runtime": "102 mins",      ← This is now a string
  "genres": [
    "drama",
    "romance",
    "war"
  ],
  "version": 1
}
```

There are a few ways we could achieve this, but a clean and simple approach is to create a custom type specifically for the `Runtime` field, and implement a `MarshalJSON()` method on this custom type.

To prevent our `internal/data/movie.go` file from getting cluttered, let's create a new file to hold the logic for the `Runtime` type:

```
$ touch internal/data/runtime.go
```

And then go ahead and add the following code:

File: internal/data/runtime.go

```
package data

import (
    "fmt"
    "strconv"
)

// Declare a custom Runtime type, which has the underlying type int32 (the same as our
// Movie struct field).
type Runtime int32

// Implement a MarshalJSON() method on the Runtime type so that it satisfies the
// json.Marshaler interface. This should return the JSON-encoded value for the movie
// runtime (in our case, it will return a string in the format "<runtime> mins").
func (r Runtime) MarshalJSON() ([]byte, error) {
    // Generate a string containing the movie runtime in the required format.
    jsonValue := fmt.Sprintf("%d mins", r)

    // Use the strconv.Quote() function on the string to wrap it in double quotes. It
    // needs to be surrounded by double quotes in order to be a valid *JSON string*.
    quotedJSONValue := strconv.Quote(jsonValue)

    // Convert the quoted string value to a byte slice and return it.
    return []byte(quotedJSONValue), nil
}
```

There are two things I'd like to emphasize here:

- If your `MarshalJSON()` method returns a JSON string value, like ours does, then you must wrap the string in double quotes before returning it. Otherwise it won't be interpreted as a *JSON string* and you'll receive a runtime error similar to this:

```
json: error calling MarshalJSON for type data.Runtime: invalid character 'm' after top-level value
```

- We're deliberately using a *value receiver* for our `MarshalJSON()` method rather than a *pointer receiver* like `func (r *Runtime) MarshalJSON()`. This gives us more flexibility because it means that our custom JSON encoding will work on *both* `Runtime` values and pointers to `Runtime` values. As [Effective Go](#) mentions:

The rule about pointers vs. values for receivers is that value methods can be invoked on pointers and values, but pointer methods can only be invoked on pointers.

Hint: If you're not confident about the difference between pointer and value receivers, then [this blog post](#) provides a good summary.

OK, now that the custom `Runtime` type is defined, open your `internal/data/movies.go` file and update the `Movie` struct to use it like so:

File: internal/data/movies.go

```
package data

import (
    "time"
)

type Movie struct {
    ID          int64    `json:"id"`
    CreatedAt   time.Time `json:"-"`
    Title       string   `json:"title"`
    Year        int32    `json:"year,omitempty"`
    // Use the Runtime type instead of int32. Note that the omitempty directive will
    // still work on this: if the Runtime field has the underlying value 0, then it will
    // be considered empty and omitted -- and the MarshalJSON() method we just made
    // won't be called at all.
    Runtime      Runtime `json:"runtime,omitempty"`
    Genres       []string `json:"genres,omitempty"`
    Version      int32    `json:"version"`
}
```

Let's try this out by restarting the API and making a request to the `GET /v1/movies/:id` endpoint. You should now see a response containing the custom runtime value in the format `"<runtime> mins"`, similar to this:

```
$ curl localhost:4000/v1/movies/123
{
  "movie": {
    "id": 123,
    "title": "Casablanca",
    "runtime": "102 mins",
    "genres": [
      "drama",
      "romance",
      "war"
    ],
    "version": 1
  }
}
```

All in all, this is a pretty nice way to generate custom JSON. Our code is succinct and clear, and we've got a custom `Runtime` type that we can use wherever and whenever we need it.

But there is a downside. It's important to be aware that using custom types can sometimes be awkward when integrating your code with other packages, and you may need to perform type conversions to change your custom type to and from a value that the other packages understand and accept.

Additional Information

There are a couple of alternative approaches that you *could* take to reach the same end result here, and I'd like to quickly describe them and talk through their pros and cons. If you're coding-along with the build, don't make any of these changes (unless you're curious, of course!).

Alternative #1 - Customizing the Movie struct

Instead of creating a custom `Runtime` type, we could have implemented a `MarshalJSON()` method on our `Movie` struct and customized the whole thing. Like this:


```
// Note that there are no struct tags on the Movie struct itself.
type Movie struct {
    ID          int64
    CreatedAt   time.Time
    Title       string
    Year        int32
    Runtime     int32
    Genres      []string
    Version     int32
}

// Implement a MarshalJSON() method on the Movie struct, so that it satisfies the
// json.Marshaler interface.
func (m Movie) MarshalJSON() ([]byte, error) {
    // Declare a variable to hold the custom runtime string (this will be the empty
    // string "" by default).
    var runtime string

    // If the value of the Runtime field is not zero, set the runtime variable to be a
    // string in the format "<runtime> mins".
    if m.Runtime != 0 {
        runtime = fmt.Sprintf("%d mins", m.Runtime)
    }

    // Create an anonymous struct to hold the data for JSON encoding. This has exactly
    // the same fields, types and tags as our Movie struct, except that the Runtime
    // field here is a string, instead of an int32. Also notice that we don't include
    // a CreatedAt field at all (there's no point including one, because we don't want
    // it to appear in the JSON output).
    aux := struct {
        ID          int64    `json:"id"`
        Title       string   `json:"title"`
        Year        int32    `json:"year,omitempty"`
        Runtime     string   `json:"runtime,omitempty"` // This is a string.
        Genres      []string `json:"genres,omitempty"`
        Version     int32    `json:"version"`
    }{
        // Set the values for the anonymous struct.
        ID:      m.ID,
        Title:   m.Title,
        Year:    m.Year,
        Runtime: runtime, // Note that we assign the value from the runtime variable here.
        Genres:  m.Genres,
        Version: m.Version,
    }

    // Encode the anonymous struct to JSON, and return it.
    return json.Marshal(aux)
}
```

Let's quickly take stock of what's going on here.

In the `MarshalJSON()` method we create a new 'anonymous' struct and assign it to the variable `aux`. This anonymous struct is basically identical to our `Movie` struct, except for the fact that the `Runtime` field has the type `string` instead of `int32`. We then copy all the values from the `Movie` struct directly into the anonymous struct, except for the `Runtime` value which we convert to a string in the format "`<runtime> mins`" first. Then finally, we *encode the anonymous struct to JSON* — not the original `Movie` struct — and return it.

It's also worth pointing out that this is designed so the `omitempty` directive still works with our custom encoding. If the value of the `Runtime` field is zero, then the local `runtime` variable will remain equal to `""`, which (as we mentioned earlier) is considered 'empty' for the purpose of encoding.

Alternative #2 - Embedding an alias

The downside of the approach above is that the code feels quite verbose and repetitive. You might be wondering: *is there a better way?*

To reduce duplication, instead of writing out all the struct fields long-hand it's possible to embed an alias of the `Movie` struct in the anonymous struct. Like so:

```
// Notice that we use the - directive on the Runtime field, so that it never appears
// in the JSON output.
type Movie struct {
    ID          int64    `json:"id"`
    CreatedAt   time.Time `json:"-"`
    Title       string    `json:"title"`
    Year        int32    `json:"year,omitempty"`
    Runtime     int32    `json:"-"`
    Genres      []string `json:"genres,omitempty"`
    Version     int32    `json:"version"`
}

func (m Movie) MarshalJSON() ([]byte, error) {
    // Create a variable holding the custom runtime string, just like before.
    var runtime string

    if m.Runtime != 0 {
        runtime = fmt.Sprintf("%d mins", m.Runtime)
    }

    // Define a MovieAlias type which has the underlying type Movie. Due to the way that
    // Go handles type definitions (https://golang.org/ref/spec#Type\_definitions) the
    // MovieAlias type will contain all the fields that our Movie struct has but,
    // importantly, none of the methods.
    type MovieAlias Movie

    // Embed the MovieAlias type inside the anonymous struct, along with a Runtime field
    // that has the type string and the necessary struct tags. It's important that we
    // embed the MovieAlias type here, rather than the Movie type directly, to avoid
    // inheriting the MarshalJSON() method of the Movie type (which would result in an
    // infinite loop during encoding).
    aux := struct {
        MovieAlias
        Runtime string `json:"runtime,omitempty"`
    }{
        MovieAlias: MovieAlias(m),
        Runtime:    runtime,
    }

    return json.Marshal(aux)
}
```

Note: Although we've used the word 'alias' here, the line `type MovieAlias Movie` is just a regular type definition. It is *not* a [type alias](#), which are generally used to help with code refactoring and migrations.

On one hand, this approach is nice because it drastically cuts the number of lines of code and reduces repetition. And if you have a large struct and only need to customize a couple of fields it can be a good option. But it's not without some downsides.

In particular:

- This technique feels like a bit of a 'trick', which hinges on the fact that newly defined types do not inherit methods. Although it's still idiomatic Go, it's more *clever* and less *clear* than the first approach. That's [not always a good trade-off](#)... especially if you have new Gophers working with you on a codebase.
- You lose granular control over the ordering of fields in the JSON response. In the above example, the `runtime` key will always now be the last item in the JSON object, like so:

```
{
  "id": 123,
  "title": "Casablanca",
  "genres": [
    "drama",
    "romance",
    "war"
  ],
  "version": 1,
  "runtime": "102 mins"
}
```

From a technical point of view, this shouldn't matter as the [JSON RFC](#) states that JSON objects are "*unordered* collections of zero or more name/value pairs". But it still might be dissatisfactory from an aesthetic or UI point of view, or be problematic if you need to maintain a precise field order for backward-compatibility purposes.

Sending Error Messages

At this point our API is sending nicely formatted JSON responses for successful requests, but if a client makes a bad request — or something goes wrong in our application — we're still sending them a plain-text error message from the `http.Error()` and `http.NotFound()` functions.

In this chapter we'll fix that by creating some additional helpers to manage errors and send the appropriate JSON responses to our clients.

If you're following along, go ahead and create a new `cmd/api/errors.go` file:

```
$ touch cmd/api/errors.go
```

And then add some helper methods like so:

File: cmd/api/errors.go

```
package main

import (
    "fmt"
    "net/http"
)

// The logError() method is a generic helper for logging an error message. Later in the
// book we'll upgrade this to use structured logging, and record additional information
// about the request including the HTTP method and URL.
func (app *application) logError(r *http.Request, err error) {
    app.logger.Println(err)
}

// The ErrorResponse() method is a generic helper for sending JSON-formatted error
// messages to the client with a given status code. Note that we're using an interface{}
// type for the message parameter, rather than just a string type, as this gives us
// more flexibility over the values that we can include in the response.
func (app *application) ErrorResponse(w http.ResponseWriter, r *http.Request, status int, message interface{}) {
    env := envelope{"error": message}

    // Write the response using the writeJSON() helper. If this happens to return an
    // error then log it, and fall back to sending the client an empty response with a
    // 500 Internal Server Error status code.
    err := app.writeJSON(w, status, env, nil)
    if err != nil {
        app.logError(r, err)
        w.WriteHeader(500)
    }
}

// The serverErrorResponse() method will be used when our application encounters an
// unexpected problem at runtime. It logs the detailed error message, then uses the
// ErrorResponse() helper to send a 500 Internal Server Error status code and JSON
// response (containing a generic error message) to the client.
func (app *application) serverErrorResponse(w http.ResponseWriter, r *http.Request, err error) {
    app.logError(r, err)

    message := "the server encountered a problem and could not process your request"
    app.errorResponse(w, r, http.StatusInternalServerError, message)
}

// The notFoundResponse() method will be used to send a 404 Not Found status code and
// JSON response to the client.
func (app *application) notFoundResponse(w http.ResponseWriter, r *http.Request) {
    message := "the requested resource could not be found"
    app.errorResponse(w, r, http.StatusNotFound, message)
}

// The methodNotAllowedResponse() method will be used to send a 405 Method Not Allowed
// status code and JSON response to the client.
func (app *application) methodNotAllowedResponse(w http.ResponseWriter, r *http.Request) {
    message := fmt.Sprintf("the %s method is not supported for this resource", r.Method)
    app.errorResponse(w, r, http.StatusMethodNotAllowed, message)
}
```

Now those are in place, let's update our API handlers to use these new helpers instead of the `http.Error()` and `http.NotFound()` functions. Like so:

File: cmd/api/healthcheck.go

```
package main

...

func (app *application) healthcheckHandler(w http.ResponseWriter, r *http.Request) {
    env := envelope{
        "status": "available",
        "system_info": map[string]string{
            "environment": app.config.env,
            "version":     version,
        },
    }

    err := app.writeJSON(w, http.StatusOK, env, nil)
    if err != nil {
        // Use the new serverErrorResponse() helper.
        app.serverErrorResponse(w, r, err)
    }
}
```

File: cmd/api/movies.go

```
package main

...

func (app *application) showMovieHandler(w http.ResponseWriter, r *http.Request) {
    id, err := app.readIDParam(r)
    if err != nil {
        // Use the new notFoundResponse() helper.
        app.notFoundResponse(w, r)
        return
    }

    movie := data.Movie{
        ID:         id,
        CreatedAt:  time.Now(),
        Title:      "Casablanca",
        Runtime:    102,
        Genres:     []string{"drama", "romance", "war"},
        Version:    1,
    }

    err = app.writeJSON(w, http.StatusOK, envelope{"movie": movie}, nil)
    if err != nil {
        // Use the new serverErrorResponse() helper.
        app.serverErrorResponse(w, r, err)
    }
}
```

Routing errors

Any error messages that our own API handlers send will now be well-formed JSON responses. Which is great!

But what about the error messages that `httprouter` automatically sends when it can't find a matching route? By default, these will still be the same plain-text (non-JSON) responses that we saw earlier in the book.

Fortunately, `httprouter` allows us to set our own custom error handlers when we initialize the router. These custom handlers must satisfy the `http.Handler` interface, which is good news for us because it means we can easily re-use the `notFoundResponse()` and `methodNotAllowedResponse()` helpers that we just made.

Open up the `cmd/api/routes.go` file and configure the `httprouter` instance like so:

```
File: cmd/api/routes.go

package main

...

func (app *application) routes() *httprouter.Router {
    router := httprouter.New()

    // Convert the notFoundResponse() helper to a http.Handler using the
    // http.HandlerFunc() adapter, and then set it as the custom error handler for 404
    // Not Found responses.
    router.NotFound = http.HandlerFunc(app.notFoundResponse)

    // Likewise, convert the methodNotAllowedResponse() helper to a http.Handler and set
    // it as the custom error handler for 405 Method Not Allowed responses.
    router.MethodNotAllowed = http.HandlerFunc(app.methodNotAllowedResponse)

    router.HandleFunc(http.MethodGet, "/v1/healthcheck", app.healthcheckHandler)
    router.HandleFunc(http.MethodPost, "/v1/movies", app.createMovieHandler)
    router.HandleFunc(http.MethodGet, "/v1/movies/:id", app.showMovieHandler)

    return router
}
```

Let's test out these changes. Restart the application, then try making some requests for endpoints that don't exist, or which use an unsupported HTTP method. You should now get some nice JSON error responses which look similar to these:

```
$ curl -i localhost:4000/foo
HTTP/1.1 404 Not Found
Content-Type: application/json
Date: Tue, 06 Apr 2021 15:13:42 GMT
Content-Length: 58

{
  "error": "the requested resource could not be found"
}

$ curl -i localhost:4000/v1/movies/abc
HTTP/1.1 404 Not Found
Content-Type: application/json
Date: Tue, 06 Apr 2021 15:14:01 GMT
Content-Length: 58

{
  "error": "the requested resource could not be found"
}

$ curl -i -X PUT localhost:4000/v1/healthcheck
HTTP/1.1 405 Method Not Allowed
Allow: GET, OPTIONS
Content-Type: application/json
Date: Tue, 06 Apr 2021 15:14:21 GMT
Content-Length: 66

{
  "error": "the PUT method is not supported for this resource"
}
```

In this final example, notice that **httprouter** still automatically sets the correct **Allow** header for us, even though it is now using our custom error handler for the response?

Additional Information

System-generated error responses

While we're on the topic of errors, I'd like to mention that in certain scenarios Go's **http.Server** may still automatically generate and send plain-text HTTP responses. These scenarios include when:

- The HTTP request specifies an unsupported HTTP protocol version.
- The HTTP request contains a missing or invalid **Host** header, or multiple **Host** headers.
- The HTTP request contains an invalid header name or value.
- The HTTP request contains an unsupported **Transfer-Encoding** header.
- The size of the HTTP request headers exceeds the server's **MaxHeaderBytes** setting.
- The client makes a HTTP request to a HTTPS server.

For example, if we try sending a request with an invalid **Host** header value we will get a response like this:

```
$ curl -i -H "Host: こんにちは" http://localhost:4000/v1/healthcheck
HTTP/1.1 400 Bad Request: malformed Host header
Content-Type: text/plain; charset=utf-8
Connection: close

400 Bad Request: malformed Host header
```

Unfortunately, these responses are hard-coded into the Go standard library, and there's nothing we can do to customize them to use JSON instead.

But while this is something to be aware of, it's not necessarily something to worry about. In a production environment it's relatively unlikely that well-behaved, non-malicious, clients would trigger these responses anyway, and we shouldn't be overly concerned if bad clients are sometimes sent a plain-text response instead of JSON.

Parsing JSON Requests

So far we've been looking at how to create and send JSON responses from our API, but in this next section of the book we're going to explore things from the other side and discuss how to *read and parse JSON requests* from clients.

To help demonstrate this, we're going to start work on the `POST /v1/movies` endpoint and `createMovieHandler` that we set up earlier.

Method	URL Pattern	Handler	Action
GET	/v1/healthcheck	healthcheckHandler	Show application information
POST	/v1/movies	createMovieHandler	Create a new movie
GET	/v1/movies/:id	showMovieHandler	Show the details of a specific movie

When a client calls this endpoint, we will expect them to provide a JSON request body containing data for the movie they want to create in our system. So, for example, if a client wanted to add a record for the movie *Moana* to our API, they would send a request body similar to this:

```
{
  "title": "Moana",
  "year": 2016,
  "runtime": 107,
  "genres": ["animation", "adventure"]
}
```

For now, we'll just focus on the reading, parsing and validation aspects of dealing with this JSON request body. Specifically you'll learn:

- How to read a request body and decode it to a native Go object using the `encoding/json` package.
- How to deal with bad requests from clients and invalid JSON, and return clear, actionable, error messages.
- How to create a reusable helper package for validating data to ensure it meets your business rules.

- Different techniques for controlling and customizing how JSON is decoded.

JSON Decoding

Just like JSON encoding, there are two approaches that you can take to *decode* JSON into a native Go object: using a `json.Decoder` type or using the `json.Unmarshal()` function.

Both approaches have their pros and cons, but for the purpose of decoding JSON from a HTTP request body, using `json.Decoder` is generally the best choice. It's more efficient than `json.Unmarshal()`, requires less code, and offers some helpful settings that you can use to tweak its behavior.

It's easiest to demonstrate how `json.Decoder` works with code, rather than words, so let's jump straight in and update our `createMovieHandler` like so:

File: cmd/api/movies.go

```
package main

import (
    "encoding/json" // New import
    "fmt"
    "net/http"
    "time"

    "greenlight.alexedwards.net/internal/data"
)

...

func (app *application) createMovieHandler(w http.ResponseWriter, r *http.Request) {
    // Declare an anonymous struct to hold the information that we expect to be in the
    // HTTP request body (note that the field names and types in the struct are a subset
    // of the Movie struct that we created earlier). This struct will be our *target
    // decode destination*.
    var input struct {
        Title   string `json:"title"`
        Year    int32  `json:"year"`
        Runtime int32  `json:"runtime"`
        Genres  []string `json:"genres"`
    }

    // Initialize a new json.Decoder instance which reads from the request body, and
    // then use the Decode() method to decode the body contents into the input struct.
    // Importantly, notice that when we call Decode() we pass a *pointer* to the input
    // struct as the target decode destination. If there was an error during decoding,
    // we also use our generic ErrorResponse() helper to send the client a 400 Bad
    // Request response containing the error message.
    err := json.NewDecoder(r.Body).Decode(&input)
    if err != nil {
        app.errorResponse(w, r, http.StatusBadRequest, err.Error())
        return
    }

    // Dump the contents of the input struct in a HTTP response.
    fmt.Fprintf(w, "%+v\n", input)
}

...
```

There are few important and interesting things about this code to point out:

- When calling `Decode()` you must pass a *non-nil pointer* as the target decode destination. If you don't use a pointer, it will return a `json.InvalidUnmarshalError` error at runtime.
- If the target decode destination is a struct — like in our case — the struct fields must be exported (start with a capital letter). Just like with encoding, they need to be exported so that they're visible to the `encoding/json` package.
- When decoding a JSON object into a struct, the key/value pairs in the JSON are mapped to the struct fields based on the struct tag names. If there is no matching struct tag, Go will attempt to decode the value into a field that matches the key name (exact matches

are preferred, but it will fall back to a case-insensitive match). *Any JSON key/value pairs which cannot be successfully mapped to the struct fields will be silently ignored.*

- There is no need to close `r.Body` after it has been read. This will be done automatically by Go's `http.Server`, so you don't have to.

OK, let's take this for a spin.

Fire up the application, then open a second terminal window and make a request to the `POST /v1/movies` endpoint with a valid JSON request body containing some movie data. You should see a response similar to this:

```
# Create a BODY variable containing the JSON data that we want to send.
$ BODY='{"title":"Moana","year":2016,"runtime":107, "genres":["animation","adventure"]}'

# Use the -d flag to send the contents of the BODY variable as the HTTP request body.
# Note that curl will default to sending a POST request when the -d flag is used.
$ curl -i -d "$BODY" localhost:4000/v1/movies
HTTP/1.1 200 OK
Date: Tue, 06 Apr 2021 17:13:46 GMT
Content-Length: 65
Content-Type: text/plain; charset=utf-8

{Title:Moana Year:2016 Runtime:107 Genres:[animation adventure]}
```

Great! That seems to have worked well. We can see from the data dumped in the response that the values we provided in the request body have been decoded into the appropriate fields of our `input` struct.

Zero values

Let's take a quick look at what happens if we omit a particular key/value pair in our JSON request body. For example, let's make a request with no `year` in the JSON, like so:

```
$ BODY='{"title":"Moana","runtime":107, "genres":["animation","adventure"]}'
$ curl -d "$BODY" localhost:4000/v1/movies
{Title:Moana Year:0 Runtime:107 Genres:[animation adventure]}
```

As you might have guessed, when we do this the `Year` field in our `input` struct is left with its zero value (which happens to be `0` because the `Year` field is an `int32` type).

This leads to an interesting question: how can you tell the difference between a client *not providing a key/value pair*, and *providing a key/value pair but deliberately setting it to its zero value*? Like this:

```
$ BODY='{"title":"Moana","year":0,"runtime":107, "genres":["animation","adventure"]}'
$ curl -d "$BODY" localhost:4000/v1/movies
{Title:Moana Year:0 Runtime:107 Genres:[animation adventure]}
```

The end result is the same, despite the different HTTP requests, and it's not immediately obvious how to tell the difference between the two scenarios. We'll circle back to this topic later in the book, but for now, it's worth just being aware of this behavior.

Additional Information

Supported destination types

It's important to mention that certain JSON types can be only be successfully decoded to certain Go types. For example, if you have the JSON string `"foo"` it can be decoded into a Go `string`, but trying to decode it into a Go `int` or `bool` will result in an error at runtime (as we'll demonstrate in the next chapter).

The following table shows the supported target decode destinations for the different JSON types:

JSON type	⇒	Supported Go types
JSON boolean	⇒	<code>bool</code>
JSON string	⇒	<code>string</code>
JSON number	⇒	<code>int*</code> , <code>uint*</code> , <code>float*</code> , <code>rune</code>
JSON array	⇒	<code>array</code> , <code>slice</code>
JSON object	⇒	<code>struct</code> , <code>map</code>

Using the `json.Unmarshal` function

As we mentioned at the start of this chapter, it's also possible to use the `json.Unmarshal()` function to decode a HTTP request body.

For example, you could use it in a handler like this:

```

func (app *application) exampleHandler(w http.ResponseWriter, r *http.Request) {
    var input struct {
        Foo string `json:"foo"`
    }

    // Use io.ReadAll() to read the entire request body into a []byte slice.
    body, err := io.ReadAll(r.Body)
    if err != nil {
        app.serverErrorResponse(w, r, err)
        return
    }

    // Use the json.Unmarshal() function to decode the JSON in the []byte slice to the
    // input struct. Again, notice that we are using a *pointer* to the input
    // struct as the decode destination.
    err = json.Unmarshal(body, &input)
    if err != nil {
        app.errorResponse(w, r, http.StatusBadRequest, err.Error())
        return
    }

    fmt.Fprintf(w, "%+v\n", input)
}

...

```

Using this approach is fine — the code works, and it’s clear and simple. But it doesn’t offer any benefits over and above the `json.Decoder` approach that we’re already taking.

Not only is the code marginally more verbose, but it’s also less efficient. If we [benchmark the relative performance](#) for this particular use case, we can see that using `json.Unmarshal()` requires about 80% more memory (B/op) than `json.Decoder`, as well as being a tiny bit slower (ns/op).

```

$ go test -run=^$ -bench=. -benchmem -count=3 -benchtime=5s
goos: linux
goarch: amd64
BenchmarkUnmarshal-8      528088      9543 ns/op    2992 B/op    20 allocs/op
BenchmarkUnmarshal-8      554365     10469 ns/op    2992 B/op    20 allocs/op
BenchmarkUnmarshal-8      537139     10531 ns/op    2992 B/op    20 allocs/op
BenchmarkDecoder-8        811063      8644 ns/op    1664 B/op    21 allocs/op
BenchmarkDecoder-8        672088      8529 ns/op    1664 B/op    21 allocs/op
BenchmarkDecoder-8       1000000      7573 ns/op    1664 B/op    21 allocs/op

```

Additional JSON decoding nuances

There are a few JSON decoding nuances that are important or interesting to know about, but which don’t fit nicely into the main content of this book. I’ve included [this appendix](#) which explains and demonstrates them in detail.

Managing Bad Requests

Our `createMovieHandler` now works well when it receives a valid JSON request body with the appropriate data. But at this point you might be wondering:

- What if the client sends something that isn't JSON, like XML or some random bytes?
- What happens if the JSON is malformed or contains an error?
- What if the JSON types don't match the types we are trying to decode into?
- What if the request doesn't even contain a body?

Well... let's look and see!

```
# Send some XML as the request body
$ curl -d '<?xml version="1.0" encoding="UTF-8"?><note><to>Alice</to></note>' localhost:4000/v1/movies
{
  "error": "invalid character '\u003c' looking for beginning of value"
}

# Send some malformed JSON (notice the trailing comma)
$ curl -d '{"title": "Moana", }' localhost:4000/v1/movies
{
  "error": "invalid character '}' looking for beginning of object key string"
}

# Send a JSON array instead of an object
$ curl -d '["foo", "bar"]' localhost:4000/v1/movies
{
  "error": "json: cannot unmarshal array into Go value of type struct { Title string
  \\"json:\\\\"title\\\\""; Year int32 \\"json:\\\\"year\\\\""; Runtime int32 \\"json:\\\\"runtime\\\\""; Genres []string \\"json:\\\\"genres\\\\""}"
}

# Send a numeric 'title' value (instead of string)
$ curl -d '{"title": 123}' localhost:4000/v1/movies
{
  "error": "json: cannot unmarshal number into Go struct field .title of type string"
}

# Send an empty request body
$ curl -X POST localhost:4000/v1/movies
{
  "error": "EOF"
}
```

In all these cases, we can see that our `createMovieHandler` is doing the right thing. When it receives an invalid request that can't be decoded into our `input` struct, no further processing takes place and the client is sent a JSON response containing the error message returned by the `Decode()` method.

For a private API which won't be used by members of the public, then this behavior is probably fine and you needn't do anything else.

But for a public-facing API, the error messages themselves aren't ideal. Some are too detailed and expose information about the underlying API implementation. Others aren't descriptive enough (like `"EOF"`), and some are just plain confusing and difficult to understand. There isn't consistency in the formatting or language used either.

To improve this, we're going to explain how to *triage the errors* returned by `Decode()` and replace them with clearer, easy-to-action, error messages to help the client debug exactly what is wrong with their JSON.

Triaging the Decode error

At this point in our application build, the `Decode()` method could potentially return the following five types of error:

Error types	Reason
<code>json.SyntaxError</code> <code>io.ErrUnexpectedEOF</code>	There is a syntax problem with the JSON being decoded.
<code>json.UnmarshalTypeError</code>	A JSON value is not appropriate for the destination Go type.
<code>json.InvalidUnmarshalError</code>	The decode destination is not valid (usually because it is not a pointer). This is actually a problem with our application code, not the JSON itself.
<code>io.EOF</code>	The JSON being decoded is empty.

Triaging these potential errors (which we can do using Go's `errors.Is()` and `errors.As()` functions) is going to make the code in our `createMovieHandler` *a lot* longer and more complicated. And the logic is something that we'll need to duplicate in other handlers throughout this project too.

So, to assist with this, let's create a new `readJSON()` helper in the `cmd/api/helpers.go` file. In this helper we'll decode the JSON from the request body as normal, then triage the errors and replace them with our own custom messages as necessary.

If you're coding-along, go ahead and add the following code to the `cmd/api/helpers.go` file:

```
File: cmd/api/helpers.go
```

```

package main

import (
    "encoding/json"
    "errors"
    "fmt" // New import
    "io"  // New import
    "net/http"
    "strconv"

    "github.com/julienschmidt/httprouter"
)

...

func (app *application) readJSON(w http.ResponseWriter, r *http.Request, dst interface{}) error {
    // Decode the request body into the target destination.
    err := json.NewDecoder(r.Body).Decode(dst)
    if err != nil {
        // If there is an error during decoding, start the triage...
        var syntaxError *json.SyntaxError
        var unmarshalTypeError *json.UnmarshalTypeError
        var invalidUnmarshalError *json.InvalidUnmarshalError

        switch {
            // Use the errors.As() function to check whether the error has the type
            // *json.SyntaxError. If it does, then return a plain-english error message
            // which includes the location of the problem.
            case errors.As(err, &syntaxError):
                return fmt.Errorf("body contains badly-formed JSON (at character %d)", syntaxError.Offset)

            // In some circumstances Decode() may also return an io.ErrUnexpectedEOF error
            // for syntax errors in the JSON. So we check for this using errors.Is() and
            // return a generic error message. There is an open issue regarding this at
            // https://github.com/golang/go/issues/25956.
            case errors.Is(err, io.ErrUnexpectedEOF):
                return errors.New("body contains badly-formed JSON")

            // Likewise, catch any *json.UnmarshalTypeError errors. These occur when the
            // JSON value is the wrong type for the target destination. If the error relates
            // to a specific field, then we include that in our error message to make it
            // easier for the client to debug.
            case errors.As(err, &unmarshalTypeError):
                if unmarshalTypeError.Field != "" {
                    return fmt.Errorf("body contains incorrect JSON type for field %q", unmarshalTypeError.Field)
                }
                return fmt.Errorf("body contains incorrect JSON type (at character %d)", unmarshalTypeError.Offset)

            // An io.EOF error will be returned by Decode() if the request body is empty. We
            // check for this with errors.Is() and return a plain-english error message
            // instead.
            case errors.Is(err, io.EOF):
                return errors.New("body must not be empty")

            // A json.InvalidUnmarshalError error will be returned if we pass a non-nil
            // pointer to Decode(). We catch this and panic, rather than returning an error
            // to our handler. At the end of this chapter we'll talk about panicking
            // versus returning errors, and discuss why it's an appropriate thing to do in
            // this specific situation.
            case errors.As(err, &invalidUnmarshalError):
                panic(err)

            // For anything else, return the error message as-is.
            default:
                return err
        }
    }
}

```

```

    }

    return nil
}

```

With this new helper in place, let's head back to the `cmd/api/movies.go` file and update our `createMovieHandler` to use it. Like so:

```

File: cmd/api/movies.go

package main

import (
    "fmt"
    "net/http"
    "time"

    "greenlight.alexedwards.net/internal/data"
)

func (app *application) createMovieHandler(w http.ResponseWriter, r *http.Request) {
    var input struct {
        Title  string `json:"title"`
        Year   int32  `json:"year"`
        Runtime int32  `json:"runtime"`
        Genres []string `json:"genres"`
    }

    // Use the new readJSON() helper to decode the request body into the input struct.
    // If this returns an error we send the client the error message along with a 400
    // Bad Request status code, just like before.
    err := app.readJSON(w, r, &input)
    if err != nil {
        app.errorResponse(w, r, http.StatusBadRequest, err.Error())
        return
    }

    fmt.Fprintf(w, "%+v\n", input)
}

...

```

Go ahead and restart the API, and then let's try this out by repeating the same bad requests that we made at the start of the chapter. You should now see our new, customized, error messages similar to this:

```
# Send some XML as the request body
$ curl -d '<?xml version="1.0" encoding="UTF-8"?><note><to>Alex</to></note>' localhost:4000/v1/movies
{
  "error": "body contains badly-formed JSON (at character 1)"
}

# Send some malformed JSON (notice the trailing comma)
$ curl -d '{"title": "Moana", }' localhost:4000/v1/movies
{
  "error": "body contains badly-formed JSON (at character 20)"
}

# Send a JSON array instead of an object
$ curl -d '["foo", "bar"]' localhost:4000/v1/movies
{
  "error": "body contains incorrect JSON type (at character 1)"
}

# Send a numeric 'title' value (instead of string)
$ curl -d '{"title": 123}' localhost:4000/v1/movies
{
  "error": "body contains incorrect JSON type for \"title\""
}

# Send an empty request body
$ curl -X POST localhost:4000/v1/movies
{
  "error": "body must not be empty"
}
```

They're looking really good. The error messages are now simpler, clearer, and consistent in their formatting, plus they don't expose any unnecessary information about our underlying program.

Feel free to play around with this if you like, and try sending different request bodies to see how the handler reacts.

Making a bad request helper

In the `createMovieHandler` code above we're using our generic `app.errorResponse()` helper to send the client a **400 Bad Request** response along with the error message.

Let's quickly replace this with a specialist `app.badRequestResponse()` helper function instead:

File: cmd/api/errors.go

```
package main

...

func (app *application) badRequestResponse(w http.ResponseWriter, r *http.Request, err error) {
    app.errorResponse(w, r, http.StatusBadRequest, err.Error())
}
```

File: cmd/api/movies.go

```
package main

...

func (app *application) createMovieHandler(w http.ResponseWriter, r *http.Request) {
    var input struct {
        Title  string `json:"title"`
        Year   int32  `json:"year"`
        Runtime int32  `json:"runtime"`
        Genres []string `json:"genres"`
    }

    err := app.readJSON(w, r, &input)
    if err != nil {
        // Use the new badRequestResponse() helper.
        app.badRequestResponse(w, r, err)
        return
    }

    fmt.Fprintf(w, "%+v\n", input)
}

...
```

This is a small change, but a useful one. As our application gradually gets more complex, using specialist helpers like this to manage different kinds of errors will help ensure that our error responses remain consistent across all our endpoints.

Additional Information

Panicking vs returning errors

The decision to panic in the `readJSON()` helper if we get a `json.InvalidUnmarshalError` error isn't taken lightly. As you're probably aware, it's generally considered best practice in Go to return your errors and handle them gracefully.

But — *in some specific circumstances* — it can be OK to panic. And you shouldn't be too

dogmatic about *not panicking* when it makes sense to.

It's helpful here to distinguish between the two classes of error that your application might encounter.

The first class of errors are *expected errors* that may occur during normal operation. Some examples of expected errors are those caused by a database query timeout, a network resource being unavailable, or bad user input. These errors don't necessarily mean there is a problem with your program itself — in fact they're often caused by things outside the control of your program. Almost all of the time it's good practice to return these kinds of errors and handle them gracefully.

The other class of errors are *unexpected errors*. These are errors which should not happen during normal operation, and if they do it is probably the result of a developer mistake or a logical error in your codebase. These errors are truly exceptional, and using panic in these circumstances is more widely accepted. In fact, the Go standard library frequently does this when you make a logical error or try to use the language features in an unintended way — such as when trying to access an out-of-bounds index in a slice, or trying to close an already-closed channel.

But even then, I'd recommend trying to return and gracefully handle unexpected errors in most cases. The exception to this is when *returning the error* adds an unacceptable amount of error handling to the rest of your codebase.

Bringing this back to our `readJSON()` helper, if we get a `json.InvalidUnmarshalError` at runtime it's because we as the developers have passed an unsupported value to `Decode()`. This is firmly an unexpected error which we *shouldn't* see under normal operation, and is something that should be picked up in development and tests long before deployment.

If we *did* return this error, rather than panicking, we would need to introduce additional code to manage it in each of our API handlers — which doesn't seem like a good trade-off for an error that we're unlikely to ever see in production.

The [Go By Example](#) page on panics summarizes all of this quite nicely:

A panic typically means something went unexpectedly wrong. Mostly we use it to fail fast on errors that shouldn't occur during normal operation and that we aren't prepared to handle gracefully.

Restricting Inputs

The changes that we made in the previous chapter to deal with invalid JSON and other bad requests were a big step in the right direction. But there are still a few things we can do to make our JSON processing even more robust.

One such thing is dealing with *unknown fields*. For example, you can try sending a request containing the unknown field `rating` to our `createMovieHandler`, like so:

```
$ curl -i -d '{"title": "Moana", "rating": "PG"}' localhost:4000/v1/movies
HTTP/1.1 200 OK
Date: Tue, 06 Apr 2021 18:51:50 GMT
Content-Length: 41
Content-Type: text/plain; charset=utf-8

{"Title:Moana Year:0 Runtime:0 Genres:[]}
```

Notice how this request works without any problems — there’s no error to inform the client that the `rating` field is not recognized by our application. In certain scenarios, silently ignoring unknown fields may be exactly the behavior you want, but in our case it would be better if we could alert the client to the issue.

Fortunately, Go’s `json.Decoder` provides a `DisallowUnknownFields()` setting that we can use to generate an error when this happens.

Another problem we have is the fact that `json.Decoder` is designed to support *streams* of JSON data. When we call `Decode()` on our request body, it actually reads the *first JSON value only* from the body and decodes it. If we made a second call to `Decode()`, it would read and decode the second value and so on.

But because we call `Decode()` once — and only once — in our `readJSON()` helper, anything after the first JSON value in the request body is ignored. This means you could send a request body containing multiple JSON values, or garbage content after the first JSON value, and our API handlers would not raise an error. For example:


```
# Body contains multiple JSON values
$ curl -i -d '{"title": "Moana"}{"title": "Top Gun"}' localhost:4000/v1/movies
HTTP/1.1 200 OK
Date: Tue, 06 Apr 2021 18:53:57 GMT
Content-Length: 41
Content-Type: text/plain; charset=utf-8

{Title:Moana Year:0 Runtime:0 Genres:[]}

# Body contains garbage content after the first JSON value
$ curl -i -d '{"title": "Moana"} :~()' localhost:4000/v1/movies
HTTP/1.1 200 OK
Date: Tue, 06 Apr 2021 18:54:15 GMT
Content-Length: 41
Content-Type: text/plain; charset=utf-8

{Title:Moana Year:0 Runtime:0 Genres:[]}
```

Again, this behavior can be very useful, but it's not the right fit for our use-case. We want requests to our `createMovieHandler` handler to contain only one single JSON object in the request body, with information about the movie to be created in our system.

To ensure that there are no additional JSON values (or any other content) in the request body, we will need to call `Decode()` a second time in our `readJSON()` helper and check that it returns an `io.EOF` (end of file) error.

Finally, there's currently no upper-limit on the maximum size of the request body that we accept. This means that our `createMovieHandler` would be a good target for any malicious clients that wish to perform a denial-of-service attack on our API. We can address this by using the `http.MaxBytesReader()` function to limit the maximum size of the request body.

Let's update our `readJSON()` helper to fix these three things:

File: cmd/api/helpers.go

```
package main

import (
    "encoding/json"
    "errors"
    "fmt"
    "io"
    "net/http"
    "strconv"
    "strings" // New import

    "github.com/julienschmidt/httprouter"
)

...

func (app *application) readJSON(w http.ResponseWriter, r *http.Request, dst interface{}) error {
    // Use http.MaxBytesReader() to limit the size of the request body to 1MB.
    maxBytes := 1_048_576
    r.Body = http.MaxBytesReader(w, r.Body, int64(maxBytes))
```

```

// Initialize the json.Decoder, and call the DisallowUnknownFields() method on it
// before decoding. This means that if the JSON from the client now includes any
// field which cannot be mapped to the target destination, the decoder will return
// an error instead of just ignoring the field.
dec := json.NewDecoder(r.Body)
dec.DisallowUnknownFields()

// Decode the request body to the destination.
err := dec.Decode(dst)
if err != nil {
    var syntaxError *json.SyntaxError
    var unmarshalTypeError *json.UnmarshalTypeError
    var invalidUnmarshalError *json.InvalidUnmarshalError

    switch {
    case errors.As(err, &syntaxError):
        return fmt.Errorf("body contains badly-formed JSON (at character %d)", syntaxError.Offset)

    case errors.Is(err, io.ErrUnexpectedEOF):
        return errors.New("body contains badly-formed JSON")

    case errors.As(err, &unmarshalTypeError):
        if unmarshalTypeError.Field != "" {
            return fmt.Errorf("body contains incorrect JSON type for field %q", unmarshalTypeError.Field)
        }
        return fmt.Errorf("body contains incorrect JSON type (at character %d)", unmarshalTypeError.Offset)

    case errors.Is(err, io.EOF):
        return errors.New("body must not be empty")

    // If the JSON contains a field which cannot be mapped to the target destination
    // then Decode() will now return an error message in the format "json: unknown
    // field "<name>". We check for this, extract the field name from the error,
    // and interpolate it into our custom error message. Note that there's an open
    // issue at https://github.com/golang/go/issues/29035 regarding turning this
    // into a distinct error type in the future.
    case strings.HasPrefix(err.Error(), "json: unknown field "):
        fieldName := strings.TrimPrefix(err.Error(), "json: unknown field ")
        return fmt.Errorf("body contains unknown key %s", fieldName)

    // If the request body exceeds 1MB in size the decode will now fail with the
    // error "http: request body too large". There is an open issue about turning
    // this into a distinct error type at https://github.com/golang/go/issues/30715.
    case err.Error() == "http: request body too large":
        return fmt.Errorf("body must not be larger than %d bytes", maxBytes)

    case errors.As(err, &invalidUnmarshalError):
        panic(err)

    default:
        return err
    }
}

// Call Decode() again, using a pointer to an empty anonymous struct as the
// destination. If the request body only contained a single JSON value this will
// return an io.EOF error. So if we get anything else, we know that there is
// additional data in the request body and we return our own custom error message.
err = dec.Decode(&struct{}{})
if err != io.EOF {
    return errors.New("body must only contain a single JSON value")
}

return nil
}

```

Once you've made those changes, let's try out the requests from earlier in the chapter again:

```
$ curl -d '{"title": "Moana", "rating": "PG"}' localhost:4000/v1/movies
{
  "error": "body contains unknown key \"rating\""
}

$ curl -d '{"title": "Moana"}{"title": "Top Gun"}' localhost:4000/v1/movies
{
  "error": "body must only contain a single JSON value"
}

$ curl -d '{"title": "Moana"} :~()' localhost:4000/v1/movies
{
  "error": "body must only contain a single JSON value"
}
```

Those are working much better now — processing of the request is terminated and the client receives a clear error message explaining exactly what the problem is.

Lastly, let's try making a request with a very large JSON body.

To demonstrate this, I've created a 1.5MB JSON file that you can download into your `/tmp` directory by running the following command:

```
$ wget -O /tmp/largefile.json https://www.alexedwards.net/static/largefile.json
```

If you try making a request to your `POST /v1/movies` endpoint with this file as the request body, the `http.MaxBytesReader()` check will kick in and you should get a response similar to this:

```
$ curl -d @/tmp/largefile.json localhost:4000/v1/movies
{
  "error": "body must not be larger than 1048576 bytes"
}
```

And with that, you'll be pleased to know that we're finally finished with the `readJSON()` helper 😊

I must admit that the code inside `readJSON()` isn't the most beautiful-looking... there's a lot of error handling and logic that we've introduced for what is ultimately a one-line call to `Decode()`. But now it's written, it's done. You don't need to touch it again, and it's something that you can copy-and-paste into other projects easily.

Custom JSON Decoding

Earlier on in this book we added some custom JSON encoding behavior to our API so that movie runtime information was displayed in the format "`<runtime> mins`" in our JSON responses.

In this chapter, we're going to look at this from the other side and update our application so that the `createMovieHandler` *accepts* runtime information in this format.

If you try sending a request with the movie runtime in this format right now, you'll get a `400 Bad Request` response (since it's not possible to decode a JSON string into an `int32` type). Like so:

```
$ curl -d '{"title": "Moana", "runtime": "107 mins"}' localhost:4000/v1/movies
{
  "error": "body contains incorrect JSON type for \"runtime\""
}
```

To make this work, what we need to do is *intercept the decoding process* and manually convert the "`<runtime> mins`" JSON string into an `int32` instead.

So how can we do that?

The `json.Unmarshaler` interface

The key thing here is knowing about Go's `json.Unmarshaler` interface, which looks like this:

```
type Unmarshaler interface {
    UnmarshalJSON([]byte) error
}
```

When Go is decoding some JSON, it will check to see if the destination type satisfies the `json.Unmarshaler` interface. If it *does* satisfy the interface, then Go will call it's `UnmarshalJSON()` method to determine how to decode the provided JSON into the target type. This is basically the reverse of the `json.Marshaler` interface that we used earlier to customize our JSON encoding behavior.

Let's take a look at how to use this in practice.

The first thing we need to do is update our `createMovieHandler` so that the `input` struct uses our custom `Runtime` type, instead of a regular `int32`. You'll remember from earlier that our `Runtime` type still has the *underlying type* `int32`, but by making this a custom type we are free to implement an `UnmarshalJSON()` method on it.

Go ahead and update the handler like so:

File: cmd/api/movies.go

```
package main

...

func (app *application) createMovieHandler(w http.ResponseWriter, r *http.Request) {
    var input struct {
        Title  string    `json:"title"`
        Year   int32     `json:"year"`
        Runtime data.Runtime `json:"runtime"` // Make this field a data.Runtime type.
        Genres []string  `json:"genres"`
    }

    err := app.readJSON(w, r, &input)
    if err != nil {
        app.badRequestResponse(w, r, err)
        return
    }

    fmt.Fprintf(w, "%+v\n", input)
}

...
```

Then let's head to the `internal/data/runtime.go` file and add a `UnmarshalJSON()` method to our `Runtime` type. In this method we need to parse the JSON string in the format `"<runtime> mins"`, convert the runtime number to an `int32`, and then assign this to the `Runtime` value itself.

It's actually a little bit intricate, and there are some important details, so it's probably best to jump into the code and explain things with comments as we go.

File: internal/data/runtime.go

```
package data

import (
    "errors" // New import
    "fmt"
    "strconv"
    "strings" // New import
)

// Define an error that our UnmarshalJSON() method can return if we're unable to parse
// or convert the JSON string successfully.
var ErrInvalidRuntimeFormat = errors.New("invalid runtime format")

type Runtime int32

...

// Implement a UnmarshalJSON() method on the Runtime type so that it satisfies the
// json.Unmarshaler interface. IMPORTANT: Because UnmarshalJSON() needs to modify the
// receiver (our Runtime type), we must use a pointer receiver for this to work
// correctly. Otherwise, we will only be modifying a copy (which is then discarded when
// this method returns).
func (r *Runtime) UnmarshalJSON(jsonValue []byte) error {
    // We expect that the incoming JSON value will be a string in the format
    // "<runtime> mins", and the first thing we need to do is remove the surrounding
    // double-quotes from this string. If we can't unquote it, then we return the
    // ErrInvalidRuntimeFormat error.
    unquotedJSONValue, err := strconv.Unquote(string(jsonValue))
    if err != nil {
        return ErrInvalidRuntimeFormat
    }

    // Split the string to isolate the part containing the number.
    parts := strings.Split(unquotedJSONValue, " ")

    // Sanity check the parts of the string to make sure it was in the expected format.
    // If it isn't, we return the ErrInvalidRuntimeFormat error again.
    if len(parts) != 2 || parts[1] != "mins" {
        return ErrInvalidRuntimeFormat
    }

    // Otherwise, parse the string containing the number into an int32. Again, if this
    // fails return the ErrInvalidRuntimeFormat error.
    i, err := strconv.ParseInt(parts[0], 10, 32)
    if err != nil {
        return ErrInvalidRuntimeFormat
    }

    // Convert the int32 to a Runtime type and assign this to the receiver. Note that we
    // use the * operator to deference the receiver (which is a pointer to a Runtime
    // type) in order to set the underlying value of the pointer.
    *r = Runtime(i)

    return nil
}
```

Once that's done, go ahead and restart the application, then make a request using the new format `runtime` value in the JSON. You should see that the request completes successfully, and the number is extracted from the string and assigned the `Runtime` field of our `input`

struct. Like so:

```
$ curl -d '{"title": "Moana", "runtime": "107 mins"}' localhost:4000/v1/movies  
{Title:Moana Year:0 Runtime:107 Genres:[]}
```

Whereas if you make the request using a JSON number, or any other format, you should now get an error response containing the message from the `ErrInvalidRuntimeFormat` variable, similar to this:

```
$ curl -d '{"title": "Moana", "runtime": 107}' localhost:4000/v1/movies  
{  
  "error": "invalid runtime format"  
}  
  
$ curl -d '{"title": "Moana", "runtime": "107 minutes"}' localhost:4000/v1/movies  
{  
  "error": "invalid runtime format"  
}
```

Validating JSON Input

In many cases, you'll want to perform additional validation checks on the data from a client to make sure it meets your specific business rules before processing it. In this chapter we'll illustrate how to do that in the context of a JSON API by updating our `createMovieHandler` to check that:

- The movie title provided by the client is not empty and is not more than 500 bytes long.
- The movie year is not empty and is between 1888 and the current year.
- The movie runtime is not empty and is a positive integer.
- The movie has between one and five (unique) genres.

If any of those checks fail, we want to send the client a `422 Unprocessable Entity` response along with error messages which clearly describe the validation failures.

Creating a validator package

To help us with validation throughout this project, we're going to create a small `internal/validator` package with some simple reusable helper types and functions. If you're coding-along, go ahead and create the following directory and file on your machine:

```
$ mkdir internal/validator
$ touch internal/validator/validator.go
```

Then in this new `internal/validator/validator.go` file add the following code:

File: internal/validator/validator.go

```
package validator

import (
    "regexp"
)

// Declare a regular expression for sanity checking the format of email addresses (we'll
// use this later in the book). If you're interested, this regular expression pattern is
// taken from https://html.spec.whatwg.org/#valid-e-mail-address. Note: if you're
// reading this in PDF or EPUB format and cannot see the full pattern, please see the
// note further down the page.
var (
    EmailRX = regexp.MustCompile(`^[a-zA-Z0-9.!#$%&'*\+\-/?^_`{ }~]+@[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?(?:\.[a-z
```



```

// Define a new Validator type which contains a map of validation errors.
type Validator struct {
    Errors map[string]string
}

// New is a helper which creates a new Validator instance with an empty errors map.
func New() *Validator {
    return &Validator{Errors: make(map[string]string)}
}

// Valid returns true if the errors map doesn't contain any entries.
func (v *Validator) Valid() bool {
    return len(v.Errors) == 0
}

// AddError adds an error message to the map (so long as no entry already exists for
// the given key).
func (v *Validator) AddError(key, message string) {
    if _, exists := v.Errors[key]; !exists {
        v.Errors[key] = message
    }
}

// Check adds an error message to the map only if a validation check is not 'ok'.
func (v *Validator) Check(ok bool, key, message string) {
    if !ok {
        v.AddError(key, message)
    }
}

// In returns true if a specific value is in a list of strings.
func In(value string, list ...string) bool {
    for i := range list {
        if value == list[i] {
            return true
        }
    }
    return false
}

// Matches returns true if a string value matches a specific regexp pattern.
func Matches(value string, rx *regexp.Regexp) bool {
    return rx.MatchString(value)
}

// Unique returns true if all string values in a slice are unique.
func Unique(values []string) bool {
    uniqueValues := make(map[string]bool)

    for _, value := range values {
        uniqueValues[value] = true
    }

    return len(values) == len(uniqueValues)
}

```

To summarize this:

In the code above we've defined a custom **Validator** type which contains a map of errors. The **Validator** type provides a **Check()** method for conditionally adding errors to the map, and a **Valid()** method which returns whether the errors map is empty or not. We've also

added `In()`, `Matches()` and `Unique()` functions to help us perform some specific validation checks.

Conceptually this `Validator` type is quite basic, but that's not a bad thing. As we'll see over the course of this book, it's surprisingly powerful in practice and gives us a lot of flexibility and control over validation checks and how we perform them.

If you're reading this book in PDF or EPUB format and you can't see the full `EmailRX` regexp pattern in the code snippet above, here it is broken up into multiple lines:

```
"^[a-zA-Z0-9.!#$%&'*\+\-\\/=?^_`{|}~]+@[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?(?:\\.[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?)*$"

```

In your code, this regexp pattern should all be on a single line with no whitespace.

Performing validation checks

Alright, let's start putting the `Validator` type to use!

The first thing we need to do is update our `cmd/api/errors.go` file to include a new `failedValidationResponse()` helper, which writes a `422 Unprocessable Entity` and the contents of the errors map from our new `Validator` type as a JSON response body.

File: `cmd/api/errors.go`

```
package main

...

// Note that the errors parameter here has the type map[string]string, which is exactly
// the same as the errors map contained in our Validator type.
func (app *application) failedValidationResponse(w http.ResponseWriter, r *http.Request, errors map[string]string) {
    app.errorResponse(w, r, http.StatusUnprocessableEntity, errors)
}
```

Then once that's done, head back to your `createMovieHandler` and update it to perform the necessary validation checks on the `input` struct. Like so:

File: cmd/api/movies.go

```
package main

import (
    "fmt"
    "net/http"
    "time"

    "greenlight.alexedwards.net/internal/data"
    "greenlight.alexedwards.net/internal/validator" // New import
)

func (app *application) createMovieHandler(w http.ResponseWriter, r *http.Request) {
    var input struct {
        Title   string    `json:"title"`
        Year    int32     `json:"year"`
        Runtime data.Runtime `json:"runtime"`
        Genres  []string  `json:"genres"`
    }

    err := app.readJSON(w, r, &input)
    if err != nil {
        app.badRequestResponse(w, r, err)
        return
    }

    // Initialize a new Validator instance.
    v := validator.New()

    // Use the Check() method to execute our validation checks. This will add the
    // provided key and error message to the errors map if the check does not evaluate
    // to true. For example, in the first line here we "check that the title is not
    // equal to the empty string". In the second, we "check that the length of the title
    // is less than or equal to 500 bytes" and so on.
    v.Check(input.Title != "", "title", "must be provided")
    v.Check(len(input.Title) <= 500, "title", "must not be more than 500 bytes long")

    v.Check(input.Year != 0, "year", "must be provided")
    v.Check(input.Year >= 1888, "year", "must be greater than 1888")
    v.Check(input.Year <= int32(time.Now().Year()), "year", "must not be in the future")

    v.Check(input.Runtime != 0, "runtime", "must be provided")
    v.Check(input.Runtime > 0, "runtime", "must be a positive integer")

    v.Check(input.Genres != nil, "genres", "must be provided")
    v.Check(len(input.Genres) >= 1, "genres", "must contain at least 1 genre")
    v.Check(len(input.Genres) <= 5, "genres", "must not contain more than 5 genres")
    // Note that we're using the Unique helper in the line below to check that all
    // values in the input.Genres slice are unique.
    v.Check(validator.Unique(input.Genres), "genres", "must not contain duplicate values")

    // Use the Valid() method to see if any of the checks failed. If they did, then use
    // the failedValidationResponse() helper to send a response to the client, passing
    // in the v.Errors map.
    if !v.Valid() {
        app.failedValidationResponse(w, r, v.Errors)
        return
    }

    fmt.Fprintf(w, "%+v\n", input)
}

...
```

With that done, we should be good to try this out. Restart the API, then go ahead and issue a request to the `POST /v1/movies` endpoint containing some invalid data. Similar to this:

```
$ BODY='{"title":"","year":1000,"runtime":"-123 mins","genres":["sci-fi","sci-fi"]}'
$ curl -i -d "$BODY" localhost:4000/v1/movies
HTTP/1.1 422 Unprocessable Entity
Content-Type: application/json
Date: Wed, 07 Apr 2021 10:33:57 GMT
Content-Length: 180

{
  "error": {
    "genres": "must not contain duplicate values",
    "runtime": "must be a positive integer",
    "title": "must be provided",
    "year": "must be greater than 1888"
  }
}
```

That's looking great. Our validation checks are working and preventing the request from being executed successfully — and even better — the client is getting a nicely formed JSON response with clear, informative, error messages for each problem.

You can also try sending a valid request, if you like. You should find that the checks pass successfully and the `input` struct is dumped in the HTTP response, just like before:

```
$ BODY='{"title":"Moana","year":2016,"runtime":"107 mins","genres":["animation","adventure"]}'
$ curl -i -d "$BODY" localhost:4000/v1/movies
HTTP/1.1 200 OK
Date: Wed, 07 Apr 2021 10:35:40 GMT
Content-Length: 65
Content-Type: text/plain; charset=utf-8

{Title:Moana Year:2016 Runtime:107 Genres:[animation adventure]}
```

Making validation rules reusable

In large projects it's likely that you'll want to reuse some of the same validation checks in multiple places. In our case — for example — we'll want to use many of these same checks later when a client *edits* the movie data.

To prevent duplication, we can collect the validation checks for a movie into a standalone `ValidateMovie()` function. In theory this function could live almost anywhere in our codebase — next to the handlers in the `cmd/api/movies.go` file, or possibly in the `internal/validators` package. But personally, I like to keep the validation checks close to the relevant domain type in the `internal/data` package.

If you're following along, reopen the `internal/data/movies.go` file and add a `ValidateMovie()` function containing the checks like so:

```
File: internal/data/movies.go

package data

import (
    "time"

    "greenlight.alexedwards.net/internal/validator" // New import
)

type Movie struct {
    ID          int64    `json:"id"`
    CreatedAt   time.Time `json:"-"`
    Title       string    `json:"title"`
    Year        int32     `json:"year,omitempty"`
    Runtime     Runtime   `json:"runtime,omitempty"`
    Genres       []string  `json:"genres,omitempty"`
    Version     int32     `json:"version"`
}

func ValidateMovie(v *validator.Validator, movie *Movie) {
    v.Check(movie.Title != "", "title", "must be provided")
    v.Check(len(movie.Title) <= 500, "title", "must not be more than 500 bytes long")

    v.Check(movie.Year != 0, "year", "must be provided")
    v.Check(movie.Year >= 1888, "year", "must be greater than 1888")
    v.Check(movie.Year <= int32(time.Now().Year()), "year", "must not be in the future")

    v.Check(movie.Runtime != 0, "runtime", "must be provided")
    v.Check(movie.Runtime > 0, "runtime", "must be a positive integer")

    v.Check(movie.Genres != nil, "genres", "must be provided")
    v.Check(len(movie.Genres) >= 1, "genres", "must contain at least 1 genre")
    v.Check(len(movie.Genres) <= 5, "genres", "must not contain more than 5 genres")
    v.Check(validator.Unique(movie.Genres), "genres", "must not contain duplicate values")
}
```

Important: Notice that the validation checks are now being performed *on a `Movie` struct* — not on the `input` struct in our handlers.

Once that's done, we need to head back to our `createMovieHandler` and update it to initialize a new `Movie` struct, copy across the data from our `input` struct, and then call this new validation function. Like so:

File: cmd/api/movies.go

```
package main

...

func (app *application) createMovieHandler(w http.ResponseWriter, r *http.Request) {
    var input struct {
        Title    string    `json:"title"`
        Year     int32     `json:"year"`
        Runtime  data.Runtime `json:"runtime"`
        Genres   []string  `json:"genres"`
    }

    err := app.readJSON(w, r, &input)
    if err != nil {
        app.badRequestResponse(w, r, err)
        return
    }

    // Copy the values from the input struct to a new Movie struct.
    movie := &data.Movie{
        Title:    input.Title,
        Year:     input.Year,
        Runtime:  input.Runtime,
        Genres:   input.Genres,
    }

    // Initialize a new Validator.
    v := validator.New()

    // Call the ValidateMovie() function and return a response containing the errors if
    // any of the checks fail.
    if data.ValidateMovie(v, movie); !v.Valid() {
        app.failedValidationResponse(w, r, v.Errors)
        return
    }

    fmt.Fprintf(w, "%+v\n", input)
}

...
```

When you're looking at this code, there might be a couple of questions in your head.

Firstly, you might be wondering why we're initializing the **Validator** instance in our handler and passing it to the **ValidateMovie()** function — rather than initializing it in **ValidateMovie()** and passing it back as a return value.

This is because as our application gets more complex we will need call *multiple* validation helpers from our handlers, rather than just one like we are above. So initializing the **Validator** in the handler, and then passing it around, gives us more flexibility.

You might also be wondering why we're decoding the JSON request into the **input** struct and then copying the data across, rather than just decoding into the **Movie** struct directly.

The problem with decoding directly into a **Movie** struct is that a client could provide the keys **id** and **version** in their JSON request, and the corresponding values would be decoded *without any error* into the **ID** and **Version** fields of the **Movie** struct — even though we don't want them to be. We *could* check the necessary fields in the **Movie** struct after the event to make sure that they are empty, but that feels a bit hacky, and decoding into an intermediary struct (like we are in our handler) is a cleaner, simpler, and more robust approach — albeit a little bit verbose.

OK, with those explanations out of the way, you should be able to start the application again and things should work the same as before from the client's perspective. If you make an invalid request, you should get a response containing the error messages similar to this:

```
$ BODY='{"title":"","year":1000,"runtime":"-123 mins","genres":["sci-fi","sci-fi"]}'
$ curl -i -d "$BODY" localhost:4000/v1/movies
HTTP/1.1 422 Unprocessable Entity
Content-Type: application/json
Date: Wed, 07 Apr 2021 10:51:00 GMT
Content-Length: 180

{
  "error": {
    "genres": "must not contain duplicate values",
    "runtime": "must be a positive integer",
    "title": "must be provided",
    "year": "must be greater than 1888"
  }
}
```

Feel free to play around with this, and try sending different values in the JSON until you're happy that all the validation checks are working as expected.

Database Setup and Configuration

In this next section of the book, we're going to move forward with our project build and set up a SQL database to persistently store our movie data.

Unlike the first *Let's Go* book (where we used MySQL as our database), this time we're going to use [PostgreSQL](#). It's open-source, very reliable and has some helpful modern features — including support for array and JSON data types, full-text search, and geospatial queries. We'll use a couple of these modern PostgreSQL features as we progress through our build.

In this section you'll learn:

- How to install and set up PostgreSQL on your local machine.
- How to use the [psql](#) interactive tool to create databases, PostgreSQL extensions and user accounts.
- How to initialize a database connection pool in Go and configure its settings to improve performance and stability.

Setting up PostgreSQL

Installing PostgreSQL

If you're following along, you'll need to install PostgreSQL on your computer at this point. The official PostgreSQL documentation contains comprehensive [download and installation instructions](#) for all types of operating systems, but if you're using macOS you should be able to install it with:

```
$ brew install postgresql
```

Or if you're using a Linux distribution you should be able to install it via your package manager. For example, if your OS supports the `apt` package manager (like Debian and Ubuntu does) you can install it with:

```
$ sudo apt install postgresql
```

On Windows machines you can install PostgreSQL using the [Chocolatey package manager](#) with the command:

```
> choco install postgresql
```

Connecting to the PostgreSQL interactive terminal

When PostgreSQL was installed a `psql` binary should also have been created on your computer. This contains a terminal-based front-end for working with PostgreSQL.

You can check that this is available by running the `psql --version` command from your terminal like so:

```
$ psql --version
psql (PostgreSQL) 13.1 (Ubuntu 13.1-1.pgdg20.04+1)
```

If you're not already familiar with PostgreSQL, the process for connecting to it for the first time using `psql` can be a bit unintuitive. So let's take a moment to walk through it.

When PostgreSQL is freshly installed it only has one user account: a superuser called **postgres**. In the first instance we need to connect to PostgreSQL as this superuser to do anything — and from there we can perform any setup steps that we need to, like creating a database and creating other users.

During installation, an *operating system* user named **postgres** should also have been created on your machine. On Unix-based systems you can check your **/etc/passwd** file to confirm this, like so:

```
$ cat /etc/passwd | grep 'postgres'
postgres:x:127:134:PostgreSQL administrator,,,:/var/lib/postgresql:/bin/bash
```

This is important because, by default, PostgreSQL uses an authentication scheme called *peer authentication* for any connections from the local machine. Peer authentication means that if the current operating system user's username matches a valid PostgreSQL user username, they can log in to PostgreSQL as that user with no further authentication. There are no passwords involved.

So it follows that if we switch to our *operating system* user called **postgres**, we should be able to connect to PostgreSQL using **psql** without needing any further authentication. In fact, you can do both these things in one step with the following command:

```
$ sudo -u postgres psql
psql (13.1 (Ubuntu 13.1-1.pgdg20.04+1))
Type "help" for help.

postgres=#
```

So, just to confirm, what we've done here is use the **sudo** command (*superuser do*) to run the **psql** command as the *operating system user* called **postgres**. That opens a session in the interactive terminal-based front-end, where we are authenticated as the PostgreSQL superuser called **postgres**.

If you want, you can confirm this by running a **"SELECT current_user"** query to see which PostgreSQL user you currently are:

```
postgres=# SELECT current_user;
 current_user
-----
 postgres
(1 row)
```

Creating databases, users, and extensions

While we're connected as the `postgres` superuser, let's create a new database for our project called `greenlight`, and then connect to it using the `\c` command like so:

```
postgres=# CREATE DATABASE greenlight;
CREATE DATABASE
postgres=# \c greenlight
You are now connected to database "greenlight" as user "postgres".
greenlight=#
```

Hint: In PostgreSQL the `\` character indicates a *meta command*. Some other useful meta commands are `\l` to list all databases, `\dt` to list tables, and `\du` to list users. You can also run `\?` to see the full list of available meta commands.

Now that our `greenlight` database exists and we're connected to it, there are a couple of tasks we need to complete.

The first task is to create a new `greenlight` user, without superuser permissions, which we can use to execute SQL migrations and connect to the database from our Go application. We want to set up this new user to use *password-based* authentication, instead of peer authentication.

PostgreSQL also has the concept of *extensions*, which add additional features on top of the standard functionality. A list of the extensions that ship with PostgreSQL can be found [here](#), and there are also [some others](#) that you can download separately.

In this project we're going to use the `citext` extension. This adds a case-insensitive character string type to PostgreSQL, which we will use later in the book to store user email addresses. It's important to note that extensions can only be added *by superusers, to a specific database*.

Go ahead and run the following commands to create a new `greenlight` user with a specific password and add the `citext` extension to our database:

```
greenlight=# CREATE ROLE greenlight WITH LOGIN PASSWORD 'pa55word';
CREATE ROLE
greenlight=# CREATE EXTENSION IF NOT EXISTS citext;
CREATE EXTENSION
```

Important: If you're following along, make sure to keep a mental note of the password you set for the **greenlight** user. You'll need it in the upcoming steps.

Once that's successfully done, you can type **exit** or **\q** to close the terminal-based front-end and revert to being your normal operating system user.

```
greenlight=# exit
```

Connecting as the new user

Before we go any further, let's prove to ourselves that everything is set up correctly and try connecting to the **greenlight** database as the **greenlight** user. When prompted, enter the password that you set in the step above.

```
$ psql --host=localhost --dbname=greenlight --username=greenlight
Password for user greenlight:
psql (13.1 (Ubuntu 13.1-1.pgdg20.04+1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression: off)
Type "help" for help.

greenlight=> SELECT current_user;
 current_user
-----
 greenlight
(1 row)

greenlight=> exit
```

Great! That confirms that our database and the new **greenlight** user with password credentials are working correctly, and that we're able to execute SQL statements as that user without any issues.

Additional Information

Optimizing PostgreSQL settings

The default settings that PostgreSQL ships with are quite conservative, and you can often improve the performance of your database by tweaking the values in your **postgresql.conf** file.

You can check where your `postgresql.conf` file lives with the following SQL query:

```
$ sudo -u postgres psql -c 'SHOW config_file;'
      config_file
-----
/etc/postgresql/13/main/postgresql.conf
(1 row)
```

[This article](#) provides a good introduction to some of the most important PostgreSQL settings, and guidance on what values are reasonable to use as a starting point. If you're interested in optimizing PostgreSQL, I recommend giving this a read.

Alternatively, you can use [this web-based tool](#) to generate suggested values based on your available system hardware. A nice feature of this tool is that it also outputs `ALTER SYSTEM` SQL statements, which you can run against your database to change the settings instead of altering your `postgresql.conf` file manually.

Connecting to PostgreSQL

OK, now that our new **greenlight** database is set up, let's look at how to connect to it from our Go application.

As you probably remember from *Let's Go*, to work with a SQL database we need to use a *database driver* to act as a 'middleman' between Go and the database itself. You can find a list of available drivers for PostgreSQL in the [Go wiki](#), but for our project we'll opt for the popular, reliable, and well-established **pq** package.

If you're coding-along with this book, then go ahead and use **go get** to download version **v1.10.2** of **pq** like so:

```
$ go get github.com/lib/pq@v1.10.2
go: downloading github.com/lib/pq v1.10.2
go get: added github.com/lib/pq v1.10.2
```

To connect to the database we'll also need a *data source name* (DSN), which is basically a string that contains the necessary connection parameters. The exact format of the DSN will depend on which database driver you're using (and should be described in the driver documentation), but when using **pq** you should be able to connect to your local **greenlight** database as the **greenlight** user with the following DSN:

```
postgres://greenlight:pa55word@localhost/greenlight
```

Establishing a connection pool

The code that we'll use for connecting to the **greenlight** database from our Go application is almost exactly the same as in the first *Let's Go* book. So we won't dwell on the details, and hopefully this will all feel very familiar.

At a high-level:

- We want the DSN to be configurable at runtime, so we will pass it to the application using a command-line flag rather than hard-coding it. For simplicity during development, we'll use the DSN above as the default value for the flag.
- In our **cmd/api/main.go** file we'll create a new **openDB()** helper function. In this helper

we'll use the `sql.Open()` function to establish a new `sql.DB` connection pool, then — because connections to the database are established lazily as and when needed for the first time — we will also need to use the `db.PingContext()` method to actually create a connection and verify that everything is set up correctly.

Let's head back to our `cmd/api/main.go` file and update it like so:

File: cmd/api/main.go

```
package main

import (
    "context"          // New import
    "database/sql"      // New import
    "flag"
    "fmt"
    "log"
    "net/http"
    "os"
    "time"

    // Import the pq driver so that it can register itself with the database/sql
    // package. Note that we alias this import to the blank identifier, to stop the Go
    // compiler complaining that the package isn't being used.
    _ "github.com/lib/pq"
)

const version = "1.0.0"

// Add a db struct field to hold the configuration settings for our database connection
// pool. For now this only holds the DSN, which we will read in from a command-line flag.
type config struct {
    port int
    env   string
    db    struct {
        dsn string
    }
}

type application struct {
    config config
    logger *log.Logger
}

func main() {
    var cfg config

    flag.IntVar(&cfg.port, "port", 4000, "API server port")
    flag.StringVar(&cfg.env, "env", "development", "Environment (development|staging|production)")

    // Read the DSN value from the db-dsn command-line flag into the config struct. We
    // default to using our development DSN if no flag is provided.
    flag.StringVar(&cfg.db.dsn, "db-dsn", "postgres://greenlight:pa55word@localhost/greenlight", "PostgreSQL DSN")

    flag.Parse()

    logger := log.New(os.Stdout, "", log.Ldate | log.Ltime)

    // Call the openDB() helper function (see below) to create the connection pool,
    // passing in the config struct. If this returns an error, we log it and exit the
    // application immediately.
    ..
```

```

db, err := openDB(cfg)
if err != nil {
    logger.Fatal(err)
}

// Defer a call to db.Close() so that the connection pool is closed before the
// main() function exits.
defer db.Close()

// Also log a message to say that the connection pool has been successfully
// established.
logger.Printf("database connection pool established")

app := &application{
    config: cfg,
    logger: logger,
}

srv := &http.Server{
    Addr:      fmt.Sprintf(":%d", cfg.port),
    Handler:   app.routes(),
    IdleTimeout: time.Minute,
    ReadTimeout: 10 * time.Second,
    WriteTimeout: 30 * time.Second,
}

logger.Printf("starting %s server on %s", cfg.env, srv.Addr)
// Because the err variable is now already declared in the code above, we need
// to use the = operator here, instead of the := operator.
err = srv.ListenAndServe()
logger.Fatal(err)
}

// The openDB() function returns a sql.DB connection pool.
func openDB(cfg config) (*sql.DB, error) {
    // Use sql.Open() to create an empty connection pool, using the DSN from the config
    // struct.
    db, err := sql.Open("postgres", cfg.db.dsn)
    if err != nil {
        return nil, err
    }

    // Create a context with a 5-second timeout deadline.
    ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
    defer cancel()

    // Use PingContext() to establish a new connection to the database, passing in the
    // context we created above as a parameter. If the connection couldn't be
    // established successfully within the 5 second deadline, then this will return an
    // error.
    err = db.PingContext(ctx)
    if err != nil {
        return nil, err
    }

    // Return the sql.DB connection pool.
    return db, nil
}

```


Important: We'll talk about using `context` to manage timeouts properly later in the book, so don't worry about this too much at the moment. For now, it's sufficient to know that if the `PingContext()` call could not complete successfully in 5 seconds, then it will return an error.

Once the `cmd/api/main.go` file is updated, go ahead and run the application again. You should now see a log message on startup confirming that the connection pool has been successfully established. Similar to this:

```
$ go run ./cmd/api
2021/04/07 15:56:54 database connection pool established
2021/04/07 15:56:54 starting development server on :4000
```

Note: If you receive the error message `pq: SSL is not enabled on the server` you should set your DSN to:
`postgres://greenlight:pa55word@localhost/greenlight?sslmode=disable.`

Decoupling the DSN

At the moment the default command-line flag value for our DSN is explicitly included as a string in the `cmd/api/main.go` file.

Even though the username and password in the DSN are just for the development database on your local machine, it would be preferable to not have this information hard-coded into our project files (which could be shared or distributed in the future).

So let's take some steps to decouple the DSN from our project code and instead store it as an environment variable on your local machine.

If you're following along, create a new `GREENLIGHT_DB_DSN` environment variable by adding the following line to either your `$HOME/.profile` or `$HOME/.bashrc` files:

```
File: $HOME/.profile
```

```
...
```

```
export GREENLIGHT_DB_DSN='postgres://greenlight:pa55word@localhost/greenlight'
```

Once that's done you'll need to reboot your computer, or — if that's not convenient right

now — run the **source** command on the file that you’ve just edited to effect the change. For example:

```
$ source $HOME/.profile
```

Note: Running **source** will affect *the current terminal window only*. So, if you switch to a different terminal window, you’ll need to run it again for it to take effect.

Either way, once you’ve rebooted or run **source** you should be able to see the value for the **GREENLIGHT_DB_DSN** environment variable in your terminal by running the **echo** command. Like so:

```
$ echo $GREENLIGHT_DB_DSN
postgres://greenlight:pa55word@localhost/greenlight
```

Now let’s update our **cmd/api/main.go** file to access the environment variable using the **os.Getenv()** function, and set this as the default value for our DSN command-line flag.

It’s fairly straightforward in practice:

```
File: cmd/api/main.go

package main

...

func main() {
    var cfg config

    flag.IntVar(&cfg.port, "port", 4000, "API server port")
    flag.StringVar(&cfg.env, "env", "development", "Environment (development|staging|production)")

    // Use the value of the GREENLIGHT_DB_DSN environment variable as the default value
    // for our db-dsn command-line flag.
    flag.StringVar(&cfg.db.dsn, "db-dsn", os.Getenv("GREENLIGHT_DB_DSN"), "PostgreSQL DSN")

    flag.Parse()

    ...
}
```

If you restart the application again now, you should find that it compiles correctly and works just like before.

You can also try specifying the **-help** flag when running the application. This should output

the descriptive text and default values for our three command-line flags, including the DSN value pulled through from the environment variable. Similar to this:

```
$ go run ./cmd/api -help
Usage of /tmp/go-build417842398/b001/exe/api:
  -db-dsn string
        PostgreSQL DSN (default "postgres://greenlight:pa55word@localhost/greenlight")
  -env string
        Environment (development|staging|production) (default "development")
  -port int
        API server port (default 4000)
```

Additional Information

Using the DSN with psql

A nice side effect of storing the DSN in an environment variable is that you can use it to easily connect to the **greenlight** database as the **greenlight** user, rather than specifying all the connection options manually when running **psql**. Like so:

```
$ psql $GREENLIGHT_DB_DSN
psql (13.2 (Ubuntu 13.2-1.pgdg20.04+1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression: off)
Type "help" for help.

greenlight=>
```

Configuring the Database Connection Pool

In the first *Let's Go* book we talked through the `sql.DB` connection pool at a high-level and demonstrated the core principles of how to use it. But in this chapter we're going to go in-depth — explaining how the connection pool works behind the scenes, and exploring the settings we can use to change and optimize its behavior.

Note: This chapter contains quite a lot of theory, and although it's interesting, it's not *critical* to the application build. If you find it too heavy-going, feel free to skim over it for now and then circle back to it later.

So how *does* the `sql.DB` connection pool work?

The most important thing to understand is that a `sql.DB` pool contains two types of connections — 'in-use' connections and 'idle' connections. A connection is marked as in-use when you are using it to perform a database task, such as executing a SQL statement or querying rows, and when the task is complete the connection is then marked as idle.

When you instruct Go to perform a database task, it will first check if any idle connections are available in the pool. If one is available, then Go will reuse this existing connection and mark it as in-use for the duration of the task. If there are no idle connections in the pool when you need one, then Go will create a new additional connection.

When Go reuses an idle connection from the pool, any problems with the connection are handled gracefully. Bad connections will automatically be re-tried twice before giving up, at which point Go will remove the bad connection from the pool and create a new one to carry out the task.

Configuring the pool

The connection pool has four methods that we can use to configure its behavior. Let's talk through them one-by-one.

The `SetMaxOpenConns` method

The `SetMaxOpenConns()` method allows you to set an upper `MaxOpenConns` limit on the number of ‘open’ connections (in-use + idle connections) in the pool. By default, the number of open connections is unlimited.

Important: Note that ‘open’ connections is equal to ‘in-use’ plus ‘idle’ connections — *it is not* just ‘in-use’ connections.

Broadly speaking, the higher that you set the `MaxOpenConns` limit, the more database queries can be performed concurrently and the lower the risk is that the *connection pool itself* will be a bottleneck in your application.

But leaving it unlimited isn’t necessarily the best thing to do. By default PostgreSQL has a hard limit of 100 open connections and, if this hard limit is hit under heavy load, it will cause our `pq` driver to return a "sorry, too many clients already" error.

Note: The hard limit on open connections can be changed in your `postgresql.conf` file using the `max_connections` setting.

To avoid this error, it makes sense limit the number of open connections in our pool to comfortably below 100 — leaving enough headroom for any other applications or sessions that also need to use PostgreSQL.

The other benefit of setting a `MaxOpenConns` limit is that it acts as a very rudimentary throttle, and prevents the database from being swamped by a huge number of tasks all at once.

But setting a limit comes with an important caveat. If the `MaxOpenConns` limit is reached, and all connections are in-use, then any further database tasks will be forced to wait until a connection becomes free and marked as idle. In the context of our API, the user’s HTTP request could ‘hang’ indefinitely while waiting for a free connection. So to mitigate this, it’s important to always set a timeout on database tasks using a `context.Context` object. We’ll explain how to do that later in the book.

The `SetMaxIdleConns` method

The `SetMaxIdleConns()` method sets an upper `MaxIdleConns` limit on the number of idle connections in the pool. By default, the maximum number of idle connections is 2.

In theory, allowing a higher number of idle connections in the pool will improve

performance because it makes it less likely that a new connection needs to be established from scratch — therefore helping to save resources.

But it's also important to realize that keeping an idle connection alive comes at a cost. It takes up memory which can otherwise be used for your application and database, and it's also possible that if a connection is idle for too long then it may become unusable. For example, by default MySQL will automatically close any connections which haven't been used for 8 hours.

So, potentially, setting `MaxIdleConns` too high may result in more connections becoming unusable and more resources being used than if you had a smaller idle connection pool — with fewer connections that are used more frequently. As a guideline: *you only want to keep a connection idle if you're likely to be using it again soon.*

Another thing to point out is that the `MaxIdleConns` limit should always be less than or equal to `MaxOpenConns`. Go enforces this and will automatically reduce the `MaxIdleConns` limit if necessary.

The `SetConnMaxLifetime` method

The `SetConnMaxLifetime()` method sets the `ConnMaxLifetime` limit — the maximum length of time that a connection can be reused for. By default, there's no maximum lifetime and connections will be reused forever.

If we set `ConnMaxLifetime` to one hour, for example, it means that all connections will be marked as 'expired' one hour after they were first created, and cannot be reused after they've expired. But note:

- This doesn't guarantee that a connection will exist in the pool for a whole hour; it's possible that a connection will become unusable for some reason and be automatically closed before then.
- A connection can still be in use more than one hour after being created — it just cannot *start* to be reused after that time.
- This isn't an idle timeout. The connection will expire one hour after it was first created — not one hour after it last became idle.
- Once every second Go runs a background cleanup operation to remove expired connections from the pool.

In theory, leaving `ConnMaxLifetime` unlimited (or setting a long lifetime) will help performance because it makes it less likely that new connections will need to be created from scratch. But in certain situations, it can be useful to enforce a shorter lifetime. For

example:

- If your SQL database enforces a maximum lifetime on connections, it makes sense to set `ConnMaxLifetime` to a slightly shorter value.
- To help facilitate swapping databases gracefully behind a load balancer.

If you do decide to set a `ConnMaxLifetime` on your pool, it's important to bear in mind the frequency at which connections will expire (and subsequently be recreated). For example, if you have 100 open connections in the pool and a `ConnMaxLifetime` of 1 minute, then your application can potentially kill and recreate up to 1.67 connections (on average) every second. You don't want the frequency to be so great that it ultimately hinders performance.

The `SetConnMaxIdleTime` method

The `SetConnMaxIdleTime()` method sets the `ConnMaxIdleTime` limit. This works in a very similar way to `ConnMaxLifetime`, except it sets the maximum length of time that a connection can be *idle* for before it is marked as expired. By default there's no limit.

If we set `ConnMaxIdleTime` to 1 hour, for example, any connections that have sat idle in the pool for 1 hour since last being used will be marked as expired and removed by the background cleanup operation.

This setting is really useful because it means that we can set a relatively high limit on the number of idle connections in the pool, but periodically free-up resources by removing any idle connections that we know aren't really being used anymore.

Putting it into practice

So that's a lot of information to take in... and what does it mean in practice? Let's summarize all the above into some actionable points.

1. As a rule of thumb, you should explicitly set a `MaxOpenConns` value. This should be comfortably below any hard limits on the number of connections imposed by your database and infrastructure, and you may also want to consider keeping it fairly low to act as a rudimentary throttle.

For this project we'll set a `MaxOpenConns` limit of 25 connections. I've found this to be a reasonable starting point for small-to-medium web applications and APIs, but ideally you should tweak this value for your hardware depending on the results of benchmarking and load-testing.

2. In general, higher `MaxOpenConns` and `MaxIdleConns` values will lead to better performance. But the returns are diminishing, and you should be aware that having a too-large idle connection pool (with connections that are not frequently re-used) can actually lead to reduced performance and unnecessary resource consumption.

Because `MaxIdleConns` should always be less than or equal to `MaxOpenConns`, we'll also limit `MaxIdleConns` to 25 connections for this project.

3. To mitigate the risk from point 2 above, you should generally set a `ConnMaxIdleTime` value to remove idle connections that haven't been used for a long time. In this project we'll set a `ConnMaxIdleTime` duration of 15 minutes.
4. It's probably OK to leave `ConnMaxLifetime` as unlimited, unless your database imposes a hard limit on connection lifetime, or you need it specifically to facilitate something like gracefully swapping databases. Neither of those things apply in this project, so we'll leave this as the default unlimited setting.

Configuring the connection pool

Rather than hard-coding these settings, let's update the `cmd/api/main.go` file to accept them as command line flags.

The command-line flag for the `ConnMaxIdleTime` value is particularly interesting, because we want it to convey a *duration of time* which we ultimately need to convert to a Go `time.Duration` type. There are a couple of options here:

1. We could use an integer to represent the number of seconds (or minutes) and convert this to a `time.Duration`.
2. We could use a string representing the duration — like `"5s"` (5 seconds) or `"10m"` (10 minutes) — and then parse it using the `time.ParseDuration()` function.

Both approaches would work fine, but we'll go with option 2 in this project. Go ahead and update the `cmd/api/main.go` file as follows:

File: `cmd/api/main.go`

```
package main

...

// Add maxOpenConns, maxIdleConns and maxIdleTime fields to hold the configuration
// settings for the connection pool.
type config struct {
    port int
```



```

env string
db struct {
    dsn          string
    maxOpenConns int
    maxIdleConns int
    maxIdleTime  string
}
}

...

func main() {
    var cfg config

    flag.IntVar(&cfg.port, "port", 4000, "API server port")
    flag.StringVar(&cfg.env, "env", "development", "Environment (development|staging|production)")

    flag.StringVar(&cfg.db.dsn, "db-dsn", os.Getenv("GREENLIGHT_DB_DSN"), "PostgreSQL DSN")

    // Read the connection pool settings from command-line flags into the config struct.
    // Notice the default values that we're using?
    flag.IntVar(&cfg.db.maxOpenConns, "db-max-open-conns", 25, "PostgreSQL max open connections")
    flag.IntVar(&cfg.db.maxIdleConns, "db-max-idle-conns", 25, "PostgreSQL max idle connections")
    flag.StringVar(&cfg.db.maxIdleTime, "db-max-idle-time", "15m", "PostgreSQL max connection idle time")

    flag.Parse()

    ...
}

func openDB(cfg config) (*sql.DB, error) {
    db, err := sql.Open("postgres", cfg.db.dsn)
    if err != nil {
        return nil, err
    }

    // Set the maximum number of open (in-use + idle) connections in the pool. Note that
    // passing a value less than or equal to 0 will mean there is no limit.
    db.SetMaxOpenConns(cfg.db.maxOpenConns)

    // Set the maximum number of idle connections in the pool. Again, passing a value
    // less than or equal to 0 will mean there is no limit.
    db.SetMaxIdleConns(cfg.db.maxIdleConns)

    // Use the time.ParseDuration() function to convert the idle timeout duration string
    // to a time.Duration type.
    duration, err := time.ParseDuration(cfg.db.maxIdleTime)
    if err != nil {
        return nil, err
    }

    // Set the maximum idle timeout.
    db.SetConnMaxIdleTime(duration)

    ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
    defer cancel()

    err = db.PingContext(ctx)
    if err != nil {
        return nil, err
    }

    return db, nil
}

```

If you go ahead and run the application again now, everything should still work correctly. You won't really notice any changes, and there's not much we can do to demonstrate the impact of these settings at this point.

But later in the book we'll do a bit of load testing, and explain how to monitor the state of the connection pool in real-time using the `db.Stats()` method. At that point, you'll be able to see some of the behavior that we've talked about in this chapter in action.

SQL Migrations

Let's get back to something a bit more concrete and tangible in this next section of the book, and take steps to create a `movies` table in our `greenlight` database.

To do this, we could simply use the `psql` tool again and run the necessary `CREATE TABLE` statement against our database. But instead, we're going to explore how to use *SQL migrations* to create the table (and more generally, manage database schema changes throughout the project).

You'll learn about:

- The high-level principles behind SQL migrations and why they are useful.
- How to use the command-line `migrate` tool to programmatically manage changes to your database schema.

An Overview of SQL Migrations

In case you're not familiar with the idea of SQL migrations, at a very high-level the concept works like this:

1. For every change that you want to make to your database schema (like creating a table, adding a column, or removing an unused index) you create a *pair of migration files*. One file is the 'up' migration which contains the SQL statements necessary to implement the change, and the other is a 'down' migration which contains the SQL statements to reverse (or *roll-back*) the change.
2. Each pair of migration files is numbered sequentially, usually `0001`, `0002`, `0003...` or with a [Unix timestamp](#), to indicate the order in which migrations should be applied to a database.
3. You use some kind of tool or script to execute or rollback the SQL statements in the sequential migration files against your database. The tool keeps track of which migrations have already been applied, so that only the necessary SQL statements are actually executed.

Using migrations to manage your database schema, rather than manually executing the SQL statements yourself, has a few benefits:

- The database schema (along with its evolution and changes) is completely described by the 'up' and 'down' SQL migration files. And because these are just regular files containing some SQL statements, they can be included and tracked alongside the rest of your code in a version control system.
- It's possible to replicate the current database schema precisely on another machine by running the necessary 'up' migrations. This is a big help when you need to manage and synchronize database schemas in different environments (development, testing, production, etc.).
- It's possible to roll-back database schema changes if necessary by applying the appropriate 'down' migrations.

Installing the migrate tool

To manage SQL migrations in this project we're going to use the [migrate](#) command-line tool (which itself is written in Go).

Detailed installation instructions for different operating systems [can be found here](#), but on macOS you should be able to install it with the command:

```
$ brew install golang-migrate
```

And on Linux and Windows, the easiest method is to download a [pre-built binary](#) and move it to a location on your system path. For example, on Linux:

```
$ curl -L https://github.com/golang-migrate/migrate/releases/download/v4.14.1/migrate.linux-amd64.tar.gz | tar xzv
$ mv migrate.linux-amd64 $GOPATH/bin/migrate
```

Before you continue, please check that it's available and working on your machine by trying to execute the [migrate](#) binary with the `-version` flag. It should output the current version number similar to this:

```
$ migrate -version
4.14.1
```

Working with SQL Migrations

Now that the `migrate` tool is installed, let's illustrate how to use it by creating a new `movies` table in our database.

The first thing we need to do is generate a pair of *migration files* using the `migrate create` command. If you're following along, go ahead and run the following command in your terminal:

```
$ migrate create -seq -ext=.sql -dir=./migrations create_movies_table
/home/alex/Projects/greenlight/migrations/000001_create_movies_table.up.sql
/home/alex/Projects/greenlight/migrations/000001_create_movies_table.down.sql
```

In this command:

- The `-seq` flag indicates that we want to use sequential numbering like `0001`, `0002`, ... for the migration files (instead of a Unix timestamp, which is the default).
- The `-ext` flag indicates that we want to give the migration files the extension `.sql`.
- The `-dir` flag indicates that we want to store the migration files in the `./migrations` directory (which will be created automatically if it doesn't already exist).
- The name `create_movies_table` is a descriptive label that we give the migration files to signify their contents.

If you look in your `migrations` directory, you should now see a pair of new 'up' and 'down' migration files like so:

```
./migrations/
├── 000001_create_movies_table.down.sql
└── 000001_create_movies_table.up.sql
```

At the moment these two new files are completely empty. Let's edit the 'up' migration file to contain the necessary `CREATE TABLE` statement for our `movies` table, like so:

File: migrations/000001_create_movies_table.up.sql

```
CREATE TABLE IF NOT EXISTS movies (  
  id bigserial PRIMARY KEY,  
  created_at timestamp(0) with time zone NOT NULL DEFAULT NOW(),  
  title text NOT NULL,  
  year integer NOT NULL,  
  runtime integer NOT NULL,  
  genres text[] NOT NULL,  
  version integer NOT NULL DEFAULT 1  
);
```

Notice here how the fields and types in this table are analogous to the fields and types in the `Movie` struct that we created earlier? This is important because it means we'll be able to easily map the data in each row of our `movies` table to a single `Movie` struct in our Go code.

If you're not familiar with the different PostgreSQL data types in the SQL statement above, the [official documentation](#) provides a comprehensive overview. But the most important things to point out are that:

- The `id` column has the type `bigserial`, which is a 64-bit auto-incrementing integer starting at 1. This will be the primary key for the table.
- The `genres` column has the type `text[]` which is an `array` of zero-or-more `text` values. It's important to note that arrays in PostgreSQL are themselves queryable and indexable, which is something that we'll demonstrate later in the book.
- You might remember from *Let's Go* that working with `NULL` values in Go can be awkward, and where possible it's easiest to just set `NOT NULL` constraints on every table column along with appropriate `DEFAULT` values — like we have above.
- For storing strings we're using the `text` type, instead of the alternative `varchar` or `varchar(n)` types. If you're interested, this [excellent blog post](#) explains why `text` is generally the best character type to use in PostgreSQL.

Alright, let's move on to the 'down' migration and add the SQL statements needed to reverse the 'up' migration that we just wrote.

File: migrations/000001_create_movies_table.down.sql

```
DROP TABLE IF EXISTS movies;
```

The `DROP TABLE` command in PostgreSQL always removes any indexes and constraints that exist for the target table, so this single statement is sufficient to reverse the 'up'.

Great, that's our first pair of migration files ready to go!

While we are at it, let's also create a second pair of migration files containing **CHECK** constraints to enforce some of our business rules at the database-level. Specifically, we want to make sure that the **runtime** value is always greater than zero, the **year** value is between 1888 and the current year, and the **genres** array always contains between 1 and 5 items.

Again, if you're following along, run the command below to create a second pair of migration files:

```
$ migrate create -seq -ext=.sql -dir=./migrations add_movies_check_constraints
/home/alex/Projects/greenlight/migrations/000002_add_movies_check_constraints.up.sql
/home/alex/Projects/greenlight/migrations/000002_add_movies_check_constraints.down.sql
```

And then add the following SQL statements to add and drop the **CHECK** constraints respectively:

File: migrations/000002_add_movies_check_constraints.up.sql

```
ALTER TABLE movies ADD CONSTRAINT movies_runtime_check CHECK (runtime >= 0);

ALTER TABLE movies ADD CONSTRAINT movies_year_check CHECK (year BETWEEN 1888 AND date_part('year', now()));

ALTER TABLE movies ADD CONSTRAINT genres_length_check CHECK (array_length(genres, 1) BETWEEN 1 AND 5);
```

File: migrations/000002_add_movies_check_constraints.down.sql

```
ALTER TABLE movies DROP CONSTRAINT IF EXISTS movies_runtime_check;

ALTER TABLE movies DROP CONSTRAINT IF EXISTS movies_year_check;

ALTER TABLE movies DROP CONSTRAINT IF EXISTS genres_length_check;
```

When we insert or update data in our **movies** table, if any of these checks fail our database driver will return an error similar to this:

```
pq: new row for relation "movies" violates check constraint "movies_year_check"
```

Note: It's perfectly acceptable for a single migration file to contain multiple SQL statements, as we see in the two files above. In fact, we could have just included the **CHECK** constraints along with the **CREATE TABLE** statement in the first pair of migration files, but for the purpose of this book having them in a separate second migration helps us to illustrate how the **migrate** tool works.

Executing the migrations

Alright, now we're ready to run the two 'up' migrations against our **greenlight** database.

If you're following along, go ahead and use the following command to execute the migrations, passing in the database DSN from your environment variable. If everything is set up correctly, you should see some output confirming that the migrations have been executed successfully. Similar to this:

```
$ migrate -path=./migrations -database=$GREENLIGHT_DB_DSN up
1/u create_movies_table (38.19761ms)
2/u add_movies_check_constraints (63.284269ms)
```

At this point, it's worth opening a connection to your database and listing the tables with the **\dt** meta command:

```
$ psql $GREENLIGHT_DB_DSN
Password for user greenlight:
psql (13.1 (Ubuntu 13.1-1.pgdg20.04+1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression: off)
Type "help" for help.

greenlight=> \dt
               List of relations
 Schema |      Name      | Type | Owner
-----+-----+-----+-----
 public | movies          | table | greenlight
 public | schema_migrations | table | greenlight
(2 rows)
```

You should see that the **movies** table has been created, along with a **schema_migrations** table, both of which are owned by the **greenlight** user.

The **schema_migrations** table is automatically generated by the **migrate** tool and used to keep track of which migrations have been applied. Let's take a quick look inside it:

```
greenlight=> SELECT * FROM schema_migrations;
 version | dirty
-----+-----
       2 | f
(1 row)
```

The **version** column here indicates that our migration files up to (and including) number **2** in the sequence have been executed against the database. The value of the **dirty** column is **false**, which indicates that the migration files were cleanly executed *without any errors* and the SQL statements they contain were successfully applied *in full*.

If you like, you can also run the `\d` meta command on the `movies` table to see the structure of the table and confirm that the `CHECK` constraints were created correctly. Like so:

```
greenlight-> \d movies

Table "public.movies"
  Column      |          Type          | Collation | Nullable |          Default          |
-----+-----+-----+-----+-----+
 id           | bigint                 |           | not null | nextval('movies_id_seq'::regclass) |
 created_at   | timestamp(0) with time zone |           | not null | now()                       |
 title        | text                   |           | not null |                               |
 year         | integer                 |           | not null |                               |
 runtime      | integer                 |           | not null |                               |
 genres       | text[]                  |           | not null |                               |
 version      | integer                 |           | not null | 1                           |
Indexes:
    "movies_pkey" PRIMARY KEY, btree (id)
Check constraints:
    "genres_length_check" CHECK (array_length(genres, 1) >= 1 AND array_length(genres, 1) <= 5)
    "movies_runtime_check" CHECK (runtime >= 0)
    "movies_year_check" CHECK (year >= 1888 AND year::double precision <= date_part('year'::text, now()))
```

Additional Information

Migrating to a specific version

As an alternative to looking at the `schema_migrations` table, if you want to see which migration version your database is currently on you can run the `migrate` tool's `version` command, like so:

```
$ migrate -path=./migrations -database=$EXAMPLE_DSN version
2
```

You can also migrate up or down to a specific version by using the `goto` command:

```
$ migrate -path=./migrations -database=$EXAMPLE_DSN goto 1
```

Executing down migrations

You can use the `down` command to roll-back by a specific number of migrations. For example, to rollback the *most recent migration* you would run:

```
$ migrate -path=./migrations -database=$EXAMPLE_DSN down 1
```

Personally, I generally prefer to use the **goto** command to perform roll-backs (as it's more explicit about the target version) and reserve use of the **down** command for rolling-back *all migrations*, like so:

```
$ migrate -path=./migrations -database=$EXAMPLE_DSN down
Are you sure you want to apply all down migrations? [y/N]
y
Applying all down migrations
2/d create_bar_table (39.988791ms)
1/d create_foo_table (59.460276ms)
```

Another variant of this is the **drop** command, which will remove all tables from the database including the **schema_migrations** table — but the database itself will remain, **along with anything else that has been created** like sequences and enums. Because of this, using **drop** can leave your database in a messy and unknown state, and it's generally better to stick with the **down** command if you want to roll back everything.

Fixing errors in SQL migrations

It's important to talk about what happens when you make a syntax error in your SQL migration files, because the behavior of the **migrate** tool can be a bit confusing to start with.

When you run a migration that contains an error, all SQL statements up to the erroneous one will be applied and then the **migrate** tool will exit with a message describing the error. Similar to this:

```
$ migrate -path=./migrations -database=$EXAMPLE_DSN up
1/u create_foo_table (36.6328ms)
2/u create_bar_table (71.835442ms)
error: migration failed: syntax error at end of input in line 0: CREATE TABLE (details: pq: syntax error at end of input)
```

If the migration file which failed contained multiple SQL statements, then it's possible that the migration file was *partially* applied before the error was encountered. In turn, this means that the database is in an unknown state as far as the **migrate** tool is concerned.

Accordingly, the **version** field in the **schema_migrations** field will contain the number for the *failed migration* and the **dirty** field will be set to **true**. At this point, if you run another migration (even a 'down' migration) you will get an error message similar to this:

```
Dirty database version {X}. Fix and force version.
```

What you need to do is investigate the original error and figure out if the migration file which failed was partially applied. If it was, then you need to manually roll-back the partially applied migration.

Once that's done, then you must also 'force' the **version** number in the **schema_migrations** table to the correct value. For example, to force the database version number to **1** you should use the **force** command like so:

```
$ migrate -path=./migrations -database=$EXAMPLE_DSN force 1
```

Once you force the version, the database is considered 'clean' and you should be able to run migrations again without any problem.

Remote migration files

The **migrate** tool also supports reading migration files from *remote sources* including Amazon S3 and GitHub repositories. For example:

```
$ migrate -source="s3://<bucket>/<path>" -database=$EXAMPLE_DSN up
$ migrate -source="github://owner/repo/path#ref" -database=$EXAMPLE_DSN up
$ migrate -source="github://user:personal-access-token@owner/repo/path#ref" -database=$EXAMPLE_DSN up
```

More information about this functionality and a full list of the supported remote resources can be [found here](#).

Running migrations on application startup

If you want, it is also possible to use the [golang-migrate/migrate](#) Go package (not the command-line tool) to automatically execute your database migrations on application start up.

We *won't* be using this approach in the book, so if you're following along please don't change your code. But roughly, the pattern looks like this:

```

package main

import (
    "context"
    "database/sql"
    "flag"
    "fmt"
    "log"
    "net/http"
    "os"
    "time"

    "github.com/golang-migrate/migrate/v4"           // New import
    "github.com/golang-migrate/migrate/v4/database/postgres" // New import
    _ "github.com/golang-migrate/migrate/v4/source/file"      // New import
    _ "github.com/lib/pq"
)

func main() {
    ...

    db, err := openDB(cfg)
    if err != nil {
        logger.Fatal(err)
    }
    defer db.Close()

    logger.Printf("database connection pool established")

    migrationDriver, err := postgres.WithInstance(db, &postgres.Config{})
    if err != nil {
        logger.PrintFatal(err, nil)
    }

    migrator, err := migrate.NewWithDatabaseInstance("file:///path/to/your/migrations", "postgres", migrationDriver)
    if err != nil {
        logger.PrintFatal(err, nil)
    }

    err = migrator.Up()
    if err != nil && err != migrate.ErrNoChange {
        logger.PrintFatal(err, nil)
    }

    logger.Printf("database migrations applied")

    ...
}

```

Although this works — and it might initially seem appealing — tightly coupling the execution of migrations with your application source code can potentially be limiting and problematic in the longer term.

The article [decoupling database migrations from server startup](#) provides a good discussion on this, and I recommend reading it if this is a topic that you're interested in. It's Python-focused, but don't let that put you off — the same principles apply in Go applications too.

CRUD Operations

In this next section of the book we're going to focus on building up the functionality for *creating, reading, updating and deleting* movies in our system.

We'll make quite rapid progress over the next few chapters, and by the end of this section we'll have the following API endpoints finished and working in full:

Method	URL Pattern	Handler	Action
GET	/v1/healthcheck	healthcheckHandler	Show application information
POST	/v1/movies	createMovieHandler	Create a new movie
GET	/v1/movies/:id	showMovieHandler	Show the details of a specific movie
PUT	/v1/movies/:id	updateMovieHandler	Update the details of a specific movie
DELETE	/v1/movies/:id	deleteMovieHandler	Delete a specific movie

In this section you'll learn:

- How to create a *database model* which isolates all the logic for executing SQL queries against your database.
- How to implement the four basic CRUD (create, read, update and delete) operations on a specific resource in the context of an API.

Setting up the Movie Model

In this chapter we're going to set up the skeleton code for our movie *database model*.

If you don't like the term *model* then you might want to think of this as your *data access* or *storage* layer instead. But whatever you prefer to call it, the principle is the same — it will encapsulate all the code for reading and writing movie data to and from our PostgreSQL database.

Let's head back to the `internal/data/movies.go` file and create a `MovieModel` struct type and some placeholder methods for performing basic CRUD (create, read, update and delete) actions against our `movies` database table.

File: internal/data/movies.go

```
package data

import (
    "database/sql" // New import
    "time"

    "greenlight.alexedwards.net/internal/validator"
)

...

// Define a MovieModel struct type which wraps a sql.DB connection pool.
type MovieModel struct {
    DB *sql.DB
}

// Add a placeholder method for inserting a new record in the movies table.
func (m MovieModel) Insert(movie *Movie) error {
    return nil
}

// Add a placeholder method for fetching a specific record from the movies table.
func (m MovieModel) Get(id int64) (*Movie, error) {
    return nil, nil
}

// Add a placeholder method for updating a specific record in the movies table.
func (m MovieModel) Update(movie *Movie) error {
    return nil
}

// Add a placeholder method for deleting a specific record from the movies table.
func (m MovieModel) Delete(id int64) error {
    return nil
}
```

As an additional step, we're going to wrap our `MovieModel` in a parent `Models` struct. Doing this is totally optional, but it has the benefit of giving you a convenient single 'container' which can hold and represent *all* your database models as your application grows.

If you're following along, create a new `internal/data/models.go` file and add the following code:

```
$ touch internal/data/models.go
```

File: internal/data/models.go

```
package data

import (
    "database/sql"
    "errors"
)

// Define a custom ErrRecordNotFound error. We'll return this from our Get() method when
// looking up a movie that doesn't exist in our database.
var (
    ErrRecordNotFound = errors.New("record not found")
)

// Create a Models struct which wraps the MovieModel. We'll add other models to this,
// like a UserModel and PermissionModel, as our build progresses.
type Models struct {
    Movies MovieModel
}

// For ease of use, we also add a New() method which returns a Models struct containing
// the initialized MovieModel.
func NewModels(db *sql.DB) Models {
    return Models{
        Movies: MovieModel{DB: db},
    }
}
```

And now, let's edit our `cmd/api/main.go` file so that the `Models` struct is initialized in our `main()` function, and then passed to our handlers as a dependency. Like so:

File: cmd/api/main.go

```
package main

import (
    "context"
    "database/sql"
    "flag"
    "fmt"
    "log"
    "net/http"
    "os"
    "time"
```



```

"greenlight.alexedwards.net/internal/data" // New import

_ "github.com/lib/pq"
)

...

// Add a models field to hold our new Models struct.
type application struct {
    config config
    logger *log.Logger
    models data.Models
}

func main() {
    var cfg config

    flag.IntVar(&cfg.port, "port", 4000, "API server port")
    flag.StringVar(&cfg.env, "env", "development", "Environment (development|staging|production)")

    flag.StringVar(&cfg.db.dsn, "db-dsn", os.Getenv("GREENLIGHT_DB_DSN"), "PostgreSQL DSN")

    flag.IntVar(&cfg.db.maxOpenConns, "db-max-open-conns", 25, "PostgreSQL max open connections")
    flag.IntVar(&cfg.db.maxIdleConns, "db-max-idle-conns", 25, "PostgreSQL max idle connections")
    flag.StringVar(&cfg.db.maxIdleTime, "db-max-idle-time", "15m", "PostgreSQL max connection idle time")

    flag.Parse()

    logger := log.New(os.Stdout, "", log.Ldate|log.Ltime)

    db, err := openDB(cfg)
    if err != nil {
        logger.Fatal(err)
    }

    defer db.Close()
    logger.Printf("database connection pool established")

    // Use the data.NewModels() function to initialize a Models struct, passing in the
    // connection pool as a parameter.
    app := &application{
        config: cfg,
        logger: logger,
        models: data.NewModels(db),
    }

    srv := &http.Server{
        Addr:         fmt.Sprintf(":%d", cfg.port),
        Handler:      app.routes(),
        IdleTimeout:  time.Minute,
        ReadTimeout:  10 * time.Second,
        WriteTimeout: 30 * time.Second,
    }

    logger.Printf("starting %s server on %s", cfg.env, srv.Addr)

    err = srv.ListenAndServe()
    logger.Fatal(err)
}

...

```

If you want, you can try restarting the application at this point. You should find that the

code compiles and runs successfully.

```
$ go run ./cmd/api
2021/04/07 21:01:22 database connection pool established
2021/04/07 21:01:22 starting development server on :4000
```

One of the nice things about this pattern is that the code to execute actions on our **movies** table will be very clear and readable from the perspective of our API handlers. For example, we'll be able to execute the **Insert()** method by simply writing:

```
app.models.Movies.Insert(...)
```

The general structure is also easy to extend. When we create more database models in the future, all we have to do is include them in the **Models** struct and they will automatically be available to our API handlers.

Additional Information

Mocking models

With a small tweak to this pattern, it's also possible to support mocking of your database models for the purpose of unit testing. We already talked about this in detail in *Let's Go*, so I don't want to rehash the same information and guidance again here. But as a quick example, you could create a **MockMovieModel** struct similar to this:

File: internal/data/movies.go

```
package data

...

type MockMovieModel struct{}

func (m MockMovieModel) Insert(movie *Movie) error {
    // Mock the action...
}

func (m MockMovieModel) Get(id int64) (*Movie, error) {
    // Mock the action...
}

func (m MockMovieModel) Update(movie *Movie) error {
    // Mock the action...
}

func (m MockMovieModel) Delete(id int64) error {
    // Mock the action...
}
```

And then update the `internal/data/models.go` file like so:

File: internal/data/models.go

```
package data

import (
    "database/sql"
    "errors"
)

var (
    ErrRecordNotFound = errors.New("record not found")
)

type Models struct {
    // Set the Movies field to be an interface containing the methods that both the
    // 'real' model and mock model need to support.
    Movies interface {
        Insert(movie *Movie) error
        Get(id int64) (*Movie, error)
        Update(movie *Movie) error
        Delete(id int64) error
    }
}

...

// Create a helper function which returns a Models instance containing the mock models
// only.
func NewMockModels() Models {
    return Models{
        Movies: MockMovieModel{},
    }
}
```

You can then call `NewMockModels()` whenever you need it in your unit tests in place of the 'real' `NewModels()` function.

Creating a New Movie

Let's begin with the `Insert()` method of our database model and update this to create a new record in our `movies` table. Specifically, we want it to execute the following SQL query:

```
INSERT INTO movies (title, year, runtime, genres)
VALUES ($1, $2, $3, $4)
RETURNING id, created_at, version
```

There are few things about this query which warrant a bit of explanation.

- It uses `$N` notation to represent *placeholder parameters* for the data that we want to insert in the `movies` table. As we explained in *Let's Go*, every time that you pass untrusted input data from a client to a SQL database it's important to use placeholder parameters to help prevent SQL injection attacks, unless you have a very specific reason for not using them.
- We're only inserting values for `title`, `year`, `runtime` and `genres`. The remaining columns in the `movies` table will be filled with *system-generated* values at the moment of insertion — the `id` will be an auto-incrementing integer, and the `created_at` and `version` values will default to the current time and `1` respectively.
- At the end of the query we have a `RETURNING` clause. This is a PostgreSQL-specific clause (it's not part of the SQL standard) that you can use to return values from any record that is being manipulated by an `INSERT`, `UPDATE` or `DELETE` statement. In this query we're using it to return the system-generated `id`, `created_at` and `version` values.

Executing the SQL query

Throughout this project we'll stick with using Go's `database/sql` package to execute our database queries, rather than using a third-party `ORM` or *other tool*. We talked about how to use `database/sql` and its various features, behaviors and gotchas in *Let's Go*, so hopefully this will feel familiar to you. Consider it a short refresher!

Normally, you would use Go's `Exec()` method to execute an `INSERT` statement against a database table. But because our SQL query is returning a single row of data (thanks to the `RETURNING` clause), we'll need to use the `QueryRow()` method here instead.

Head back to your `internal/data/movies.go` file and update it like so:

File: internal/data/movies.go

```
package data

import (
    "database/sql"
    "time"

    "greenlight.alexedwards.net/internal/validator"

    "github.com/lib/pq" // New import
)

...

// The Insert() method accepts a pointer to a movie struct, which should contain the
// data for the new record.
func (m MovieModel) Insert(movie *Movie) error {
    // Define the SQL query for inserting a new record in the movies table and returning
    // the system-generated data.
    query := `
        INSERT INTO movies (title, year, runtime, genres)
        VALUES ($1, $2, $3, $4)
        RETURNING id, created_at, version`

    // Create an args slice containing the values for the placeholder parameters from
    // the movie struct. Declaring this slice immediately next to our SQL query helps to
    // make it nice and clear *what values are being used where* in the query.
    args := []interface{}{movie.Title, movie.Year, movie.Runtime, pq.Array(movie.Genres)}

    // Use the QueryRow() method to execute the SQL query on our connection pool,
    // passing in the args slice as a variadic parameter and scanning the system-
    // generated id, created_at and version values into the movie struct.
    return m.DB.QueryRow(query, args...).Scan(&movie.ID, &movie.CreatedAt, &movie.Version)
}

...
```

That code is nice and succinct, but there are a few important things to mention.

Because the `Insert()` method signature takes a `*Movie` pointer as the parameter, when we call `Scan()` to read in the system-generated data we're updating the values *at the location the parameter points to*. Essentially, our `Insert()` method *mutates* the `Movie` struct that we pass to it and adds the system-generated values to it.

The next thing to talk about is the placeholder parameter inputs, which we declare in an `args` slice like this:

```
args := []interface{}{movie.Title, movie.Year, movie.Runtime, pq.Array(movie.Genres)}
```

Storing the inputs in a slice isn't strictly necessary, but as mentioned in the code comments

above it's a nice pattern that can help the clarity of your code. Personally, I usually do this for SQL queries with more than three placeholder parameters.

Also, notice the final value in the slice? In order to store our `movie.Genres` value (which is a `[]string` slice) in the database, we need to pass it through the `pq.Array()` adapter function before executing the SQL query.

Behind the scenes, the `pq.Array()` adapter takes our `[]string` slice and converts it to a `pq.StringArray` type. In turn, the `pq.StringArray` type implements the `driver.Valuer` and `sql.Scanner` interfaces necessary to translate our native `[]string` slice to and from a value that our PostgreSQL database can understand and store in a `text[]` array column.

Hint: You can also use the `pq.Array()` adapter function in the same way with `[]bool`, `[]byte`, `[]int32`, `[]int64`, `[]float32` and `[]float64` slices in your Go code.

Hooking it up to our API handler

Now for the exciting part. Let's hook up the `Insert()` method to our `createMovieHandler` so that our `POST /v1/movies` endpoint works in full. Like so:

File: cmd/api/movies.go

```
package main

...

func (app *application) createMovieHandler(w http.ResponseWriter, r *http.Request) {
    var input struct {
        Title   string    `json:"title"`
        Year    int32     `json:"year"`
        Runtime data.Runtime `json:"runtime"`
        Genres  []string  `json:"genres"`
    }

    err := app.readJSON(w, r, &input)
    if err != nil {
        app.badRequestResponse(w, r, err)
        return
    }

    // Note that the movie variable contains a *pointer* to a Movie struct.
    movie := &data.Movie{
        Title:   input.Title,
        Year:    input.Year,
        Runtime: input.Runtime,
        Genres:  input.Genres,
    }

    v := validator.New()

    if data.ValidateMovie(v, movie); !v.Valid() {
        app.failedValidationResponse(w, r, v.Errors)
        return
    }

    // Call the Insert() method on our movies model, passing in a pointer to the
    // validated movie struct. This will create a record in the database and update the
    // movie struct with the system-generated information.
    err = app.models.Movies.Insert(movie)
    if err != nil {
        app.serverErrorResponse(w, r, err)
        return
    }

    // When sending a HTTP response, we want to include a Location header to let the
    // client know which URL they can find the newly-created resource at. We make an
    // empty http.Header map and then use the Set() method to add a new Location header,
    // interpolating the system-generated ID for our new movie in the URL.
    headers := make(http.Header)
    headers.Set("Location", fmt.Sprintf("/v1/movies/%d", movie.ID))

    // Write a JSON response with a 201 Created status code, the movie data in the
    // response body, and the Location header.
    err = app.writeJSON(w, http.StatusCreated, envelope{"movie": movie}, headers)
    if err != nil {
        app.serverErrorResponse(w, r, err)
    }
}

...
```

OK, let's try this out!

Restart the API, then open up a second terminal window and make the following request to the **POST /v1/movies** endpoint:

```
$ BODY='{"title":"Moana","year":2016,"runtime":"107 mins", "genres":["animation","adventure"]}'
$ curl -i -d "$BODY" localhost:4000/v1/movies
HTTP/1.1 201 Created
Content-Type: application/json
Location: /v1/movies/1
Date: Wed, 07 Apr 2021 19:21:41 GMT
Content-Length: 156

{
  "movie": {
    "id": 1,
    "title": "Moana",
    "year": 2016,
    "runtime": "107 mins",
    "genres": [
      "animation",
      "adventure"
    ],
    "version": 1
  }
}
```

That's looking perfect. We can see that the JSON response contains all the information for the new movie, including the system-generated ID and version numbers. And the response also includes the **Location: /v1/movies/1** header, pointing to the URL which will later represent the movie in our system.

Creating additional records

While we're at it, let's create a few more records in the system to help us demonstrate different functionality as our build progresses.

If you're coding-along, please run the following commands to create three more movie records in the database:

```

$ BODY='{"title":"Black Panther","year":2018,"runtime":"134 mins","genres":["action","adventure"]}'
$ curl -d "$BODY" localhost:4000/v1/movies
{
  "movie": {
    "id": 2,
    "title": "Black Panther",
    "year": 2018,
    "runtime": "134 mins",
    "genres": [
      "action",
      "adventure"
    ],
    "version": 1
  }
}

$ BODY='{"title":"Deadpool","year":2016, "runtime":"108 mins","genres":["action","comedy"]}'
$ curl -d "$BODY" localhost:4000/v1/movies
{
  "movie": {
    "id": 3,
    "title": "Deadpool",
    "year": 2016,
    "runtime": "108 mins",
    "genres": [
      "action",
      "comedy"
    ],
    "version": 1
  }
}

$ BODY='{"title":"The Breakfast Club","year":1986, "runtime":"96 mins","genres":["drama"]}'
$ curl -d "$BODY" localhost:4000/v1/movies
{
  "movie": {
    "id": 4,
    "title": "The Breakfast Club",
    "year": 1986,
    "runtime": "96 mins",
    "genres": [
      "drama"
    ],
    "version": 1
  }
}

```

At this point you might also want to take a look in PostgreSQL to confirm that the records have been created properly. You should see that the contents of the **movies** table now looks similar to this (including the appropriate movie **genres** in an array).

```
$ psql $GREENLIGHT_DB_DSN
Password for user greenlight:
psql (13.1 (Ubuntu 13.1-1.pgdg20.04+1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression: off)
Type "help" for help.
```

```
greenlight=> SELECT * FROM movies;
 id |      created_at      |      title      | year | runtime |      genres      | version
-----+-----+-----+-----+-----+-----+-----
  1 | 2021-04-07 21:21:41+02 | Moana           | 2016 |    107 | {animation,adventure} |      1
  2 | 2021-04-07 21:28:28+02 | Black Panther   | 2018 |    134 | {action,adventure}   |      1
  3 | 2021-04-07 21:28:36+02 | Deadpool        | 2016 |    108 | {action,comedy}      |      1
  4 | 2021-04-07 21:28:44+02 | The Breakfast Club | 1986 |     96 | {drama}              |      1
(4 rows)
```

Additional Information

\$N notation

A nice feature of the PostgreSQL placeholder parameter **\$N** notation is that you can use the same parameter value in multiple places in your SQL statement. For example, it's perfectly acceptable to write code like this:

```
// This SQL statement uses the $1 parameter twice, and the value `123` will be used in
// both locations where $1 appears.
stmt := "UPDATE foo SET bar = $1 + $2 WHERE bar = $1"
err := db.Exec(stmt, 123, 456)
if err != nil {
    ...
}
```

Executing multiple statements

Occasionally you might find yourself in the position where you want to execute more than one SQL statement in the same database call, like this:

```
stmt := `
    UPDATE foo SET bar = true;
    UPDATE foo SET baz = false;`

err := db.Exec(stmt)
if err != nil {
    ...
}
```

Having multiple statements in the same call is supported by the **pq** driver, *so long as the*

statements do not contain any placeholder parameters. If they do contain placeholder parameters, then you'll receive the following error message at runtime:

```
pq: cannot insert multiple commands into a prepared statement
```

To work around this, you will need to either split out the statements into separate database calls, or if that's not possible, you can create a [custom function](#) in PostgreSQL which acts as a wrapper around the multiple SQL statements that you want to run.

Fetching a Movie

Now let's move on to the code for *fetching and displaying the data for a specific movie*.

Again, we'll begin in our database model here, and start by updating the `Get()` method to execute the following SQL query:

```
SELECT id, created_at, title, year, runtime, genres, version
FROM movies
WHERE id = $1
```

Because our `movies` table uses the `id` column as its primary key, this query will only ever return exactly one database row (or none at all). So, it's appropriate for us to execute this query using Go's `QueryRow()` method again.

If you're following along, open up your `internal/data/movies.go` file and update it like so:

File: internal/data/movies.go

```
package data

import (
    "database/sql"
    "errors" // New import
    "time"

    "greenlight.alexedwards.net/internal/validator"

    "github.com/lib/pq"
)

...

func (m MovieModel) Get(id int64) (*Movie, error) {
    // The PostgreSQL bigserial type that we're using for the movie ID starts
    // auto-incrementing at 1 by default, so we know that no movies will have ID values
    // less than that. To avoid making an unnecessary database call, we take a shortcut
    // and return an ErrRecordNotFound error straight away.
    if id < 1 {
        return nil, ErrRecordNotFound
    }

    // Define the SQL query for retrieving the movie data.
    query := `
        SELECT id, created_at, title, year, runtime, genres, version
        FROM movies
        WHERE id = $1`

    // Declare a Movie struct to hold the data returned by the query.
    var movie Movie
```

```
// Execute the query using the QueryRow() method, passing in the provided id value
// as a placeholder parameter, and scan the response data into the fields of the
// Movie struct. Importantly, notice that we need to convert the scan target for the
// genres column using the pq.Array() adapter function again.
err := m.DB.QueryRow(query, id).Scan(
    &movie.ID,
    &movie.CreatedAt,
    &movie.Title,
    &movie.Year,
    &movie.Runtime,
    pq.Array(&movie.Genres),
    &movie.Version,
)

// Handle any errors. If there was no matching movie found, Scan() will return
// a sql.ErrNoRows error. We check for this and return our custom ErrRecordNotFound
// error instead.
if err != nil {
    switch {
    case errors.Is(err, sql.ErrNoRows):
        return nil, ErrRecordNotFound
    default:
        return nil, err
    }
}

// Otherwise, return a pointer to the Movie struct.
return &movie, nil
}

...
```

Hopefully the code above should feel clear and familiar — it’s a straight lift of the pattern that we discussed in detail in *Let’s Go*.

The only real thing of note is the fact that we need to use the `pq.Array()` adapter again when scanning in the genres data from the PostgreSQL `text[]` array. If we didn’t use this adapter, we would get the following error at runtime:

```
sql: Scan error on column index 5, name "genres": unsupported Scan, storing driver.Value type []uint8 into type *[]string
```

Updating the API handler

OK, the next thing we need to do is update our `showMovieHandler` so that it calls the `Get()` method we just made. The handler should check to see if `Get()` returns an `ErrRecordNotFound` error — and if it does, the client should be sent a `404 Not Found` response. Otherwise, we can go ahead and render the returned `Movie` struct in a JSON response.

Like so:

File: cmd/api/movies.go

```
package main

import (
    "errors" // New import
    "fmt"
    "net/http"

    "greenlight.alexedwards.net/internal/data"
    "greenlight.alexedwards.net/internal/validator"
)

...

func (app *application) showMovieHandler(w http.ResponseWriter, r *http.Request) {
    id, err := app.readIDParam(r)
    if err != nil {
        app.notFoundResponse(w, r)
        return
    }

    // Call the Get() method to fetch the data for a specific movie. We also need to
    // use the errors.Is() function to check if it returns a data.ErrRecordNotFound
    // error, in which case we send a 404 Not Found response to the client.
    movie, err := app.models.Movies.Get(id)
    if err != nil {
        switch {
            case errors.Is(err, data.ErrRecordNotFound):
                app.notFoundResponse(w, r)
            default:
                app.serverErrorResponse(w, r, err)
        }
        return
    }

    err = app.writeJSON(w, http.StatusOK, envelope{"movie": movie}, nil)
    if err != nil {
        app.serverErrorResponse(w, r, err)
    }
}
```

Great! That's all nice and succinct, thanks to the structure and helpers that we've already put in place.

Feel free to give this a try by restarting the API and looking up a movie that you've already created in the database. For example:

```
$ curl -i localhost:4000/v1/movies/2
HTTP/1.1 200 OK
Content-Type: application/json
Date: Wed, 07 Apr 2021 19:37:12 GMT
Content-Length: 161

{
  "movie": {
    "id": 2,
    "title": "Black Panther",
    "year": 2018,
    "runtime": "134 mins",
    "genres": [
      "action",
      "adventure"
    ],
    "version": 1
  }
}
```

And likewise, you can also try making a request with a movie ID that doesn't exist in the database yet (but is otherwise valid). In that scenario you should receive a **404 Not Found** response like so:

```
$ curl -i localhost:4000/v1/movies/42
HTTP/1.1 404 Not Found
Content-Type: application/json
Date: Wed, 07 Apr 2021 19:37:58 GMT
Content-Length: 58

{
  "error": "the requested resource could not be found"
}
```

Additional Information

Why not use an unsigned integer for the movie ID?

At the start of the **Get()** method we have the following code which checks if the movie **id** parameter is less than 1:

```
func (m MovieModel) Get(id int64) (*Movie, error) {
    if id < 1 {
        return nil, ErrRecordNotFound
    }

    ...
}
```


This might have led you to wonder: *if the movie ID is never negative, why aren't we using an unsigned `uint64` type to store the ID in our Go code, instead of an `int64`?*

There are two reasons for this.

The first reason is because *PostgreSQL doesn't have unsigned integers*. It's generally sensible to align your Go and database integer types to avoid overflows or other compatibility problems, so because PostgreSQL doesn't have unsigned integers, this means that we should avoid using `uint*` types in our Go code for any values that we're reading/writing to PostgreSQL too.

Instead, it's best to align the integer types based on the following table:

PostgreSQL type	Go type
<code>smallint</code> , <code>smallserial</code>	<code>int16</code> (-32768 to 32767)
<code>integer</code> , <code>serial</code>	<code>int32</code> (-2147483648 to 2147483647)
<code>bigint</code> , <code>bigserial</code>	<code>int64</code> (-9223372036854775808 to 9223372036854775807)

There's also another, more subtle, reason. Go's `database/sql` package *doesn't actually support* any integer values greater than 9223372036854775807 (the maximum value for an `int64`). It's possible that a `uint64` value could be greater than this, which would in turn lead to Go generating a runtime error similar to this:

```
sql: converting argument $1 type: uint64 values with high bit set are not supported
```

By sticking with an `int64` in our Go code, we eliminate the risk of ever encountering this error.

Updating a Movie

In this chapter we'll continue building up our application and add a brand-new endpoint which allows clients to *update the data for a specific movie*.

Method	URL Pattern	Handler	Action
GET	/v1/healthcheck	healthcheckHandler	Show application information
POST	/v1/movies	createMovieHandler	Create a new movie
GET	/v1/movies/:id	showMovieHandler	Show the details of a specific movie
PUT	/v1/movies/:id	updateMovieHandler	Update the details of a specific movie

More precisely, we'll set up the endpoint so that a client can edit the **title**, **year**, **runtime** and **genres** values for a movie. In our project the **id** and **created_at** values should never change once they've been created, and the **version** value isn't something that the client should control, so we won't allow those fields to be edited.

For now, we'll configure this endpoint so that it performs a *complete replacement* of the values for a movie. This means that the client will need to provide values for *all* editable fields in their JSON request body... even if they only want to change one of them.

For example, if a client wanted to add the genre **sci-fi** to the movie *Black Panther* in our database, they would need to send a JSON request body which looks like this:

```
{
  "title": "Black Panther",
  "year": 2018,
  "runtime": "134 mins",
  "genres": [
    "action",
    "adventure",
    "sci-fi"
  ]
}
```

Executing the SQL query

Let's start in our database model again, and edit the **Update()** method to execute the

following SQL query:

```
UPDATE movies
SET title = $1, year = $2, runtime = $3, genres = $4, version = version + 1
WHERE id = $5
RETURNING version
```

Notice here that we're incrementing the **version** value as part of the query? And then at the end we're using the **RETURNING** clause to return this new, incremented, **version** value.

Like before, this query returns a single row of data so we'll also need to use Go's **QueryRow()** method to execute it. If you're following along, head back to your **internal/data/movies.go** file and fill in the **Update()** method like so:

```
File: internal/data/movies.go

package data

...

func (m MovieModel) Update(movie *Movie) error {
    // Declare the SQL query for updating the record and returning the new version
    // number.
    query := `
        UPDATE movies
        SET title = $1, year = $2, runtime = $3, genres = $4, version = version + 1
        WHERE id = $5
        RETURNING version`

    // Create an args slice containing the values for the placeholder parameters.
    args := []interface{}{
        movie.Title,
        movie.Year,
        movie.Runtime,
        pq.Array(movie.Genres),
        movie.ID,
    }

    // Use the QueryRow() method to execute the query, passing in the args slice as a
    // variadic parameter and scanning the new version value into the movie struct.
    return m.DB.QueryRow(query, args...).Scan(&movie.Version)
}

...
```

It's important to emphasize that — just like our **Insert()** method — the **Update()** method takes a pointer to a **Movie** struct as the input parameter and mutates it in-place again — this time updating it with the new version number only.

Creating the API handler

Now let's head back to our `cmd/api/movies.go` file and update it to include the brand-new `updateMovieHandler` method.

Method	URL Pattern	Handler	Action
PUT	<code>/v1/movies/:id</code>	<code>updateMovieHandler</code>	Update the details of a specific movie

The nice thing about this handler is that we've already laid all the groundwork for it — our work here is mainly just a case of linking up the code and helper functions that we've already written to handle the request.

Specifically, we'll need to:

1. Extract the movie ID from the URL using the `app.readIDParam()` helper.
2. Fetch the corresponding movie record from the database using the `Get()` method that we made in the previous chapter.
3. Read the JSON request body containing the updated movie data into an `input` struct.
4. Copy the data across from the `input` struct to the movie record.
5. Check that the updated movie record is valid using the `data.ValidateMovie()` function.
6. Call the `Update()` method to store the updated movie record in our database.
7. Write the updated movie data in a JSON response using the `app.writeJSON()` helper.

So let's go ahead and do exactly that:

File: `cmd/api/movies.go`

```
package main

...

func (app *application) updateMovieHandler(w http.ResponseWriter, r *http.Request) {
    // Extract the movie ID from the URL.
    id, err := app.readIDParam(r)
    if err != nil {
        app.notFoundResponse(w, r)
        return
    }

    // Fetch the existing movie record from the database, sending a 404 Not Found
    // response to the client if we couldn't find a matching record.
    movie, err := app.models.Movies.Get(id)
    if err != nil {
        switch {
            case errors.Is(err, data.ErrRecordNotFound):
                app.notFoundResponse(w, r)
            default:
                app.serverErrorResponse(w, r, err)
        }
    }
    return
}
```

```

}

// Declare an input struct to hold the expected data from the client.
var input struct {
    Title    string    `json:"title"`
    Year     int32     `json:"year"`
    Runtime  data.Runtime `json:"runtime"`
    Genres   []string   `json:"genres"`
}

// Read the JSON request body data into the input struct.
err = app.readJSON(w, r, &input)
if err != nil {
    app.badRequestResponse(w, r, err)
    return
}

// Copy the values from the request body to the appropriate fields of the movie
// record.
movie.Title = input.Title
movie.Year = input.Year
movie.Runtime = input.Runtime
movie.Genres = input.Genres

// Validate the updated movie record, sending the client a 422 Unprocessable Entity
// response if any checks fail.
v := validator.New()

if data.ValidateMovie(v, movie); !v.Valid() {
    app.failedValidationResponse(w, r, v.Errors)
    return
}

// Pass the updated movie record to our new Update() method.
err = app.models.Movies.Update(movie)
if err != nil {
    app.serverErrorResponse(w, r, err)
    return
}

// Write the updated movie record in a JSON response.
err = app.writeJSON(w, http.StatusOK, envelope{"movie": movie}, nil)
if err != nil {
    app.serverErrorResponse(w, r, err)
}
}

```

Lastly, to finish this off, we also need to update our application routes to include the new endpoint. Like so:

File: cmd/api/routes.go

```
package main

...

func (app *application) routes() *httprouter.Router {
    router := httprouter.New()

    router.NotFound = http.HandlerFunc(app.notFoundResponse)
    router.MethodNotAllowed = http.HandlerFunc(app.methodNotAllowedResponse)

    router.HandleFunc(http.MethodGet, "/v1/healthcheck", app.healthcheckHandler)
    router.HandleFunc(http.MethodPost, "/v1/movies", app.createMovieHandler)
    router.HandleFunc(http.MethodGet, "/v1/movies/:id", app.showMovieHandler)
    // Add the route for the PUT /v1/movies/:id endpoint.
    router.HandleFunc(http.MethodPut, "/v1/movies/:id", app.updateMovieHandler)

    return router
}
```

Using the new endpoint

And with that, we're now ready to try this out!

To demonstrate, let's continue with the example we gave at the start of this chapter and update our record for *Black Panther* to include the genre **sci-fi**. As a reminder, the record currently looks like this:

```
$ curl localhost:4000/v1/movies/2
{
  "movie": {
    "id": 2,
    "title": "Black Panther",
    "year": 2018,
    "runtime": "134 mins",
    "genres": [
      "action",
      "adventure"
    ],
    "version": 1
  }
}
```

To make the update to the genres field we can execute the following API call:

```
$ BODY='{"title":"Black Panther","year":2018,"runtime":"134 mins","genres":["sci-fi","action","adventure"]}'
$ curl -X PUT -d "$BODY" localhost:4000/v1/movies/2
{
  "movie": {
    "id": 2,
    "title": "Black Panther",
    "year": 2018,
    "runtime": "134 mins",
    "genres": [
      "sci-fi",
      "action",
      "adventure"
    ],
    "version": 2
  }
}
```

That's looking great — we can see from the response that the movie genres have been updated to include "sci-fi", and the version number has been incremented to 2 like we would expect.

You should also be able to verify that the change has been persisted by making a GET /v1/movies/2 request again, like so:

```
$ curl localhost:4000/v1/movies/2
{
  "movie": {
    "id": 2,
    "title": "Black Panther",
    "year": 2018,
    "runtime": "134 mins",
    "genres": [
      "sci-fi",
      "action",
      "adventure"
    ],
    "version": 2
  }
}
```

Deleting a Movie

In this chapter we'll add our final CRUD endpoint so that a client can *delete a specific movie* from our system.

Method	URL Pattern	Handler	Action
GET	/v1/healthcheck	healthcheckHandler	Show application information
POST	/v1/movies	createMovieHandler	Create a new movie
GET	/v1/movies/:id	showMovieHandler	Show the details of a specific movie
PUT	/v1/movies/:id	updateMovieHandler	Update the details of a specific movie
DELETE	/v1/movies/:id	deleteMovieHandler	Delete a specific movie

Compared to the other endpoints in our API, the behavior that we want to implement here is quite straightforward.

- If a movie with the **id** provided in the URL exists in the database, we want to delete the corresponding record and return a success message to the client.
- If the movie **id** doesn't exist, we want to return a **404 Not Found** response to the client.

The SQL query to delete the record in our database is also simple:

```
DELETE FROM movies
WHERE id = $1
```

In this case the SQL query returns no rows, so it's appropriate for us to use Go's `Exec()` method to execute it.

One of the nice things about `Exec()` is that it returns a `sql.Result` object, which contains information about the *number of rows that the query affected*. In our scenario here, this is really useful information.

- If the number of rows affected is **1**, then we know that the movie existed in the table and has now been deleted... so we can send the client a success message.
- Conversely, if the number of rows affected is **0** we know that no movie with that **id** existed at the point we tried to delete it, and we can send the client a **404 Not Found**

response.

Adding the new endpoint

Let's go ahead and update the `Delete()` method in our database model. Essentially, we want this to execute the SQL query above and return an `ErrRecordNotFound` error if the number of rows affected is `0`. Like so:

```
File: internal/data/movies.go

package data

...

func (m MovieModel) Delete(id int64) error {
    // Return an ErrRecordNotFound error if the movie ID is less than 1.
    if id < 1 {
        return ErrRecordNotFound
    }

    // Construct the SQL query to delete the record.
    query := `
        DELETE FROM movies
        WHERE id = $1`

    // Execute the SQL query using the Exec() method, passing in the id variable as
    // the value for the placeholder parameter. The Exec() method returns a sql.Result
    // object.
    result, err := m.DB.Exec(query, id)
    if err != nil {
        return err
    }

    // Call the RowsAffected() method on the sql.Result object to get the number of rows
    // affected by the query.
    rowsAffected, err := result.RowsAffected()
    if err != nil {
        return err
    }

    // If no rows were affected, we know that the movies table didn't contain a record
    // with the provided ID at the moment we tried to delete it. In that case we
    // return an ErrRecordNotFound error.
    if rowsAffected == 0 {
        return ErrRecordNotFound
    }

    return nil
}
```

Once that's done, let's head to our `cmd/api/movies.go` file and add a new `deleteMovieHandler` method. In this we need to read the movie ID from the request URL, call the `Delete()` method that we just made, and — based on the return value from `Delete()` — send the appropriate response to the client.

File: cmd/api/movies.go

```
package main

...

func (app *application) deleteMovieHandler(w http.ResponseWriter, r *http.Request) {
    // Extract the movie ID from the URL.
    id, err := app.readIDParam(r)
    if err != nil {
        app.notFoundResponse(w, r)
        return
    }

    // Delete the movie from the database, sending a 404 Not Found response to the
    // client if there isn't a matching record.
    err = app.models.Movies.Delete(id)
    if err != nil {
        switch {
        case errors.Is(err, data.ErrRecordNotFound):
            app.notFoundResponse(w, r)
        default:
            app.serverErrorResponse(w, r, err)
        }
        return
    }

    // Return a 200 OK status code along with a success message.
    err = app.writeJSON(w, http.StatusOK, envelope{"message": "movie successfully deleted"}, nil)
    if err != nil {
        app.serverErrorResponse(w, r, err)
    }
}
```

Note: You may prefer to send an empty response body and a **204 No Content** status code here, rather than a "movie successfully deleted" message. It really depends on who your clients are — if they are humans, then sending a message similar to the above is a nice UX touch; if they are machines, then a **204 No Content** response is probably sufficient.

Finally, we need to hook up the new route in our `cmd/api/routes.go` file:

File: cmd/api/routes.go

```
package main

...

func (app *application) routes() *httprouter.Router {
    router := httprouter.New()

    router.NotFound = http.HandlerFunc(app.notFoundResponse)
    router.MethodNotAllowed = http.HandlerFunc(app.methodNotAllowedResponse)

    router.HandlerFunc(http.MethodGet, "/v1/healthcheck", app.healthcheckHandler)
    router.HandlerFunc(http.MethodPost, "/v1/movies", app.createMovieHandler)
    router.HandlerFunc(http.MethodGet, "/v1/movies/:id", app.showMovieHandler)
    router.HandlerFunc(http.MethodPatch, "/v1/movies/:id", app.updateMovieHandler)
    // Add the route for the DELETE /v1/movies/:id endpoint.
    router.HandlerFunc(http.MethodDelete, "/v1/movies/:id", app.deleteMovieHandler)

    return router
}
```

OK, let's restart the API and try this out by deleting *Deadpool* from our movie database (this should have an ID of **3** if you've been following along). The delete operation should work without any problems, and you should receive the confirmation message like so:

```
$ curl -X DELETE localhost:4000/v1/movies/3
{
  "message": "movie successfully deleted"
}
```

If you repeat the same request to delete the already-deleted movie, you should now get a **404 Not Found** response and error message. Similar to this:

```
$ curl -X DELETE localhost:4000/v1/movies/3
{
  "error": "the requested resource could not be found"
}
```

Advanced CRUD Operations

In this section of the book we're going to look at a few more 'advanced' patterns that you might want to use for the CRUD endpoints that your API provides.

You'll learn:

- How to support *partial* updates to a resource (so that the client only needs to send the data that they want to change).
- How to use *optimistic concurrency control* to avoid race conditions when two clients try to update the same resource at the same time.
- How to use *context timeouts* to terminate long-running database queries and prevent unnecessary resource use.

Handling Partial Updates

In this chapter we're going to change the behavior of the `updateMovieHandler` so that it supports *partial updates* of the movie records. Conceptually this is a little more complicated than making a *complete replacement*, which is why we laid the groundwork with that approach first.

As an example, let's say that we notice that the release year for *The Breakfast Club* is wrong in our database (it should actually be 1985, not 1986). It would be nice if we could send a JSON request containing only the change that needs to be applied, instead of all the movie data, like so:

```
{"year": 1985}
```

Let's quickly look at what happens if we try to send this request right now:

```
$ curl -X PUT -d '{"year": 1985}' localhost:4000/v1/movies/4
{
  "error": {
    "genres": "must be provided",
    "runtime": "must be provided",
    "title": "must be provided"
  }
}
```

As we mentioned earlier in the book, when decoding the request body any fields in our `input` struct which *don't* have a corresponding JSON key/value pair will retain their *zero-value*. We happen to check for these zero-values during validation and return the error messages that you see above.

In the context of partial update this causes a problem. How do we tell the difference between:

- A client *providing a key/value pair which has a zero-value value* — like `{"title": ""}` — in which case we want to return a validation error.
- A client *not providing a key/value pair* in their JSON at all — in which case we want to 'skip' updating the field but not send a validation error.

To help answer this, let's quickly remind ourselves of what the zero-values are for different Go types.

Go type	Zero-value
<code>int*</code> , <code>uint*</code> , <code>float*</code> , <code>complex</code>	<code>0</code>
<code>string</code>	<code>""</code>
<code>bool</code>	<code>false</code>
<code>func</code> , <code>array</code> , <code>slice</code> , <code>map</code> , <code>chan</code> and <code>pointers</code>	<code>nil</code>

The key thing to notice here is that *pointers have the zero-value `nil`*.

So — in theory — we could change the fields in our `input` struct to be pointers. Then to see if a client has provided a particular key/value pair in the JSON, we can simply check whether the corresponding field in the `input` struct equals `nil` or not.

```
// Use pointers for the Title, Year and Runtime fields.
var input struct {
    Title    *string    `json:"title"` // This will be nil if there is no corresponding key in the JSON.
    Year     *int32     `json:"year"`  // Likewise...
    Runtime  *data.Runtime `json:"runtime"` // Likewise...
    Genres   []string   `json:"genres"` // We don't need to change this because slices already have the zero-value nil.
}
```

Performing the partial update

Let's put this into practice, and edit our `updateMovieHandler` method so it supports partial updates as follows:

```
File: cmd/api/movies.go

package main

...

func (app *application) updateMovieHandler(w http.ResponseWriter, r *http.Request) {
    id, err := app.readIDParam(r)
    if err != nil {
        app.notFoundResponse(w, r)
        return
    }

    // Retrieve the movie record as normal.
    movie, err := app.models.Movies.Get(id)
    if err != nil {
        switch {
        case errors.Is(err, data.ErrRecordNotFound):
            app.notFoundResponse(w, r)
        default:
            app.serverErrorResponse(w, r, err)
        }
    }
}
```

```

    return
}

// Use pointers for the Title, Year and Runtime fields.
var input struct {
    Title    *string    `json:"title"`
    Year      *int32     `json:"year"`
    Runtime  *data.Runtime `json:"runtime"`
    Genres   []string  `json:"genres"`
}

// Decode the JSON as normal.
err = app.readJSON(w, r, &input)
if err != nil {
    app.badRequestResponse(w, r, err)
    return
}

// If the input.Title value is nil then we know that no corresponding "title" key/
// value pair was provided in the JSON request body. So we move on and leave the
// movie record unchanged. Otherwise, we update the movie record with the new title
// value. Importantly, because input.Title is a now a pointer to a string, we need
// to dereference the pointer using the * operator to get the underlying value
// before assigning it to our movie record.
if input.Title != nil {
    movie.Title = *input.Title
}

// We also do the same for the other fields in the input struct.
if input.Year != nil {
    movie.Year = *input.Year
}
if input.Runtime != nil {
    movie.Runtime = *input.Runtime
}
if input.Genres != nil {
    movie.Genres = input.Genres // Note that we don't need to dereference a slice.
}

v := validator.New()

if data.ValidateMovie(v, movie); !v.Valid() {
    app.failedValidationResponse(w, r, v.Errors)
    return
}

err = app.models.Movies.Update(movie)
if err != nil {
    app.serverErrorResponse(w, r, err)
    return
}

err = app.writeJSON(w, http.StatusOK, envelope{"movie": movie}, nil)
if err != nil {
    app.serverErrorResponse(w, r, err)
}
}

```

To summarize this: we've changed our `input` struct so that all the fields now have the zero-value `nil`. After parsing the JSON request, we then go through the `input` struct fields and only update the movie record if the new value is *not* `nil`.

In addition to this, for API endpoints which perform *partial updates* on a resource, it's appropriate to use the HTTP method **PATCH** rather than **PUT** (which is intended for replacing a resource in full).

So, before we try out our new code, let's quickly update our `cmd/api/routes.go` file so that our `updateMovieHandler` is only used for **PATCH** requests.

File: cmd/api/routes.go

```
package main

...

func (app *application) routes() *httprouter.Router {
    router := httprouter.New()

    router.NotFound = http.HandlerFunc(app.notFoundResponse)
    router.MethodNotAllowed = http.HandlerFunc(app.methodNotAllowedResponse)

    router.HandleFunc(http.MethodGet, "/v1/healthcheck", app.healthcheckHandler)
    router.HandleFunc(http.MethodPost, "/v1/movies", app.createMovieHandler)
    router.HandleFunc(http.MethodGet, "/v1/movies/:id", app.showMovieHandler)
    // Require a PATCH request, rather than PUT.
    router.HandleFunc(http.MethodPatch, "/v1/movies/:id", app.updateMovieHandler)
    router.HandleFunc(http.MethodDelete, "/v1/movies/:id", app.deleteMovieHandler)

    return router
}
```

Demonstration

With that all set up, let's check that this partial update functionality works by correcting the release year for *The Breakfast Club* to 1985. Like so:

```
$ curl -X PATCH -d '{"year": 1985}' localhost:4000/v1/movies/4
{
  "movie": {
    "id": 4,
    "title": "The Breakfast Club",
    "year": 1985,
    "runtime": "96 mins",
    "genres": [
      "drama"
    ],
    "version": 2
  }
}
```

Again, that's looking good. We can see that the **year** value has been correctly updated, and the **version** number has been incremented, but none of the other data fields have been changed.

Let's also quickly try the same request but including an empty `title` value. In this case the update will be blocked and you should receive a validation error, like so:

```
$ curl -X PATCH -d '{"year": 1985, "title": ""}' localhost:4000/v1/movies/4
{
  "error": {
    "title": "must be provided"
  }
}
```

Additional Information

Null values in JSON

One special-case to be aware of is when the client explicitly *supplies a field in the JSON request with the value `null`*. In this case, our handler will ignore the field and treat it like it hasn't been supplied.

For example, the following request would result in no changes to the movie record (apart from the `version` number being incremented):

```
$ curl -X PATCH -d '{"title": null, "year": null}' localhost:4000/v1/movies/4
{
  "movie": {
    "id": 4,
    "title": "The Breakfast Club",
    "year": 1985,
    "runtime": "96 mins",
    "genres": [
      "drama"
    ],
    "version": 3
  }
}
```

In an ideal world this type of request would return some kind of validation error. But — unless you write your own custom JSON parser — there is no way to determine the difference between the client not supplying a key/value pair in the JSON, or supplying it with the value `null`.

In most cases, it will probably suffice to explain this special-case behavior in client documentation for the endpoint and say something like “*JSON items with `null` values will be ignored and will remain unchanged*”.

Optimistic Concurrency Control

The eagle-eyed of you might have noticed a small problem in our `updateMovieHandler` — there is a `race condition` if two clients try to update the same movie record at exactly the same time.

To illustrate this, let's pretend that we have two clients using our API: Alice and Bob. Alice wants to correct the runtime value for *The Breakfast Club* to 97 mins, and Bob wants to add the genre 'comedy' to the same movie.

Now imagine that Alice and Bob send these two update requests at *exactly* the same time. As we explained in *Let's Go*, Go's `http.Server` handles each HTTP request in its own goroutine, so when this happens the code in our `updateMovieHandler` will be running concurrently in two different goroutines.

Let's step through what could potentially happen in this scenario:

1. Alice's goroutine calls `app.models.Movies.Get()` to retrieve a copy of the movie record (which has version number `N`).
2. Bob's goroutine calls `app.models.Movies.Get()` to retrieve a copy of the movie record (which still has version `N`).
3. Alice's goroutine changes the runtime to 97 minutes in its copy of the movie record.
4. Bob's goroutine updates the genres to include 'comedy' in its copy of the movie record.
5. Alice's goroutine calls `app.models.Movies.Update()` with its copy of the movie record. The movie record is written to the database and the version number is incremented to `N+1`.
6. Bob's goroutine calls `app.models.Movies.Update()` with its copy of the movie record. The movie record is written to the database and the version number is incremented to `N+2`.

Despite making two separate updates, only Bob's update will be reflected in the database at the end because the two goroutines were *racing* each other to make the change. Alice's update to the movie runtime will be lost when Bob's update overwrites it with the old runtime value. And this happens silently — there's nothing to inform either Alice or Bob of the problem.

Note: This specific type of race condition is known as a *data race*. Data races can occur when two or more goroutines try to use a piece of shared data (in this example the movie record) at the same time, but the result of their operations is dependent on the exact order that the scheduler executes their instructions.

Preventing the data race

Now we understand that the data race exists and why it's happening, *how can we prevent it?*

There are a couple of options, but the simplest and cleanest approach in this case is to use a form of **optimistic locking** based on the **version** number in our movie record.

The fix works like this:

1. Alice and Bob's goroutines both call `app.models.Movies.Get()` to retrieve a copy of the movie record. Both of these records have the version number **N**.
2. Alice and Bob's goroutines make their respective changes to the movie record.
3. Alice and Bob's goroutines call `app.models.Movies.Update()` with their copies of the movie record. But the update is only executed *if the version number in the database is still N*. If it has changed, then we don't execute the update and send the client an error message instead.

This means that the first update request that reaches our database will succeed, and whoever is making the second update will receive an error message instead of having their change applied.

To make this work, we'll need to change the SQL statement for updating a movie so that it looks like this:

```
UPDATE movies
SET title = $1, year = $2, runtime = $3, genres = $4, version = version + 1
WHERE id = $5 AND version = $6
RETURNING version
```

Notice that in the **WHERE** clause we're now looking for a record with a specific ID *and a specific version number*?

If no matching record can be found, this query will result in a `sql.ErrNoRows` error and we know that the version number has been changed (or the record has been deleted completely). Either way, it's a form of *edit conflict* and we can use this as a trigger to send

the client an appropriate error response.

Implementing optimistic locking

OK, that's enough theory... let's put this into practice!

We'll start by creating a custom `ErrEditConflict` error that we can return from our database models in the event of a conflict. We'll use this later in the book when working with user records too, so it makes sense to define it in the `internal/data/models.go` file like so:

File: internal/data/models.go

```
package data

import (
    "database/sql"
    "errors"
)

var (
    ErrRecordNotFound = errors.New("record not found")
    ErrEditConflict   = errors.New("edit conflict")
)

...
```

Next let's update our database model's `Update()` method to execute the new SQL query and manage the situation where a matching record couldn't be found.

File: internal/data/movies.go

```
package data

...

func (m MovieModel) Update(movie *Movie) error {
    // Add the 'AND version = $6' clause to the SQL query.
    query := `
        UPDATE movies
        SET title = $1, year = $2, runtime = $3, genres = $4, version = version + 1
        WHERE id = $5 AND version = $6
        RETURNING version`

    args := []interface{}{
        movie.Title,
        movie.Year,
        movie.Runtime,
        pq.Array(movie.Genres),
        movie.ID,
        movie.Version, // Add the expected movie version.
    }

    // Execute the SQL query. If no matching row could be found, we know the movie
    // version has changed (or the record has been deleted) and we return our custom
    // ErrEditConflict error.
    err := m.DB.QueryRow(query, args...).Scan(&movie.Version)
    if err != nil {
        switch {
        case errors.Is(err, sql.ErrNoRows):
            return ErrEditConflict
        default:
            return err
        }
    }

    return nil
}

...
```

Next let's head to our `cmd/api/errors.go` file and create a new `editConflictResponse()` helper. We want this to send a **409 Conflict** response, along with a plain-English error message that explains the problem to the client.

File: cmd/api/errors.go

```
package main

...

func (app *application) editConflictResponse(w http.ResponseWriter, r *http.Request) {
    message := "unable to update the record due to an edit conflict, please try again"
    app.errorResponse(w, r, http.StatusConflict, message)
}
```

And then as the final step, we need to change our `updateMovieHandler` so that it checks for

an `ErrEditConflict` error and calls the `editConflictResponse()` helper if necessary. Like so:

```
File: cmd/api/movies.go

package main

...

func (app *application) updateMovieHandler(w http.ResponseWriter, r *http.Request) {

    ...

    // Intercept any ErrEditConflict error and call the new editConflictResponse()
    // helper.
    err = app.models.Movies.Update(movie)
    if err != nil {
        switch {
        case errors.Is(err, data.ErrEditConflict):
            app.editConflictResponse(w, r)
        default:
            app.serverErrorResponse(w, r, err)
        }
        return
    }

    err = app.writeJSON(w, http.StatusOK, envelope{"movie": movie}, nil)
    if err != nil {
        app.serverErrorResponse(w, r, err)
    }
}
```

At this point, our `updateMovieHandler` should now be safe from the race condition that we've been talking about. If two goroutines are executing the code at the same time, the first update will succeed, and the second will fail because the `version` number in the database no longer matches the expected value.

Let's try this out by using the `xargs` command to send a bunch of concurrent requests to our endpoint. Assuming that your computer executes the requests closely-enough together, you should find that some requests succeed but the others now fail with a `409 Conflict` status code. Like so:

```
$ xargs -I % -P8 curl -X PATCH -d '{"runtime": "97 mins"}' "localhost:4000/v1/movies/4" < <(printf '%s\n' {1..8})
{
  "movie": {
    "id": 4,
    "title": "Breakfast Club",
    "year": 1985,
    "runtime": "97 mins",
    "genres": [
      "drama"
    ],
    "version": 4
  }
}
{
  "error": "unable to update the record due to an edit conflict, please try again"
}
{
  "error": "unable to update the record due to an edit conflict, please try again"
}
{
  "error": "unable to update the record due to an edit conflict, please try again"
}
{
  "error": "unable to update the record due to an edit conflict, please try again"
}
{
  "error": "unable to update the record due to an edit conflict, please try again"
}
{
  "error": "unable to update the record due to an edit conflict, please try again"
}
{
  "movie": {
    "id": 4,
    "title": "Breakfast Club",
    "year": 1985,
    "runtime": "97 mins",
    "genres": [
      "drama"
    ],
    "version": 5
  }
}
}
```

Bringing this to a close, the race condition that we've been demonstrating in this chapter is fairly innocuous. But in other applications this exact class of race condition can have much more serious consequences — such as when updating the stock level for a product in an online store, or updating the balance of an account.

As I mentioned briefly in *Let's Go*, it's good to get into the habit of thinking about race conditions whenever you write code, and structure your applications to either manage them or avoid them completely — no matter how innocuous they might seem at the time of development.

Additional Information

Round-trip locking

One of the nice things about the optimistic locking pattern that we've used here is that you can extend it so the client passes the version number that *they expect* in an **If-Not-Match** or **X-Expected-Version** header.

In certain applications, this can be useful to help the client ensure they are not sending *their update request* based on outdated information.

Very roughly, you could implement this by adding a check to your **updateMovieHandler** like so:

```
func (app *application) updateMovieHandler(w http.ResponseWriter, r *http.Request) {
    id, err := app.readIDParam(r)
    if err != nil {
        app.notFoundResponse(w, r)
        return
    }

    movie, err := app.models.Movies.Get(id)
    if err != nil {
        switch {
        case errors.Is(err, data.ErrRecordNotFound):
            app.notFoundResponse(w, r)
        default:
            app.serverErrorResponse(w, r, err)
        }
        return
    }

    // If the request contains a X-Expected-Version header, verify that the movie
    // version in the database matches the expected version specified in the header.
    if r.Header.Get("X-Expected-Version") != "" {
        if strconv.FormatInt(int64(movie.Version), 32) != r.Header.Get("X-Expected-Version") {
            app.editConflictResponse(w, r)
            return
        }
    }

    ...
}
```

Locking on other fields or types

Using an incrementing integer **version** number as the basis for an optimistic lock is safe and computationally cheap. I'd recommend using this approach unless you have a specific reason not to.

As an alternative, you could use a **last_updated** timestamp as the basis for the lock. But this

is less safe — there's the theoretical possibility that two clients could update a record at exactly the same time, and using a timestamp also introduces the risk of further problems if your server's clock is wrong or becomes wrong over time.

If it's important to you that the version identifier isn't guessable, then a good option is to use a high-entropy random string such as a UUID in the **version** field. PostgreSQL has a **UUID type** and the **uuid-oss** extension which you could use for this purpose like so:

```
UPDATE movies
SET title = $1, year = $2, runtime = $3, genres = $4, version = uuid_generate_v4()
WHERE id = $5 AND version = $6
RETURNING version
```

Managing SQL Query Timeouts

So far in our code we've been using Go's `Exec()` and `QueryRow()` methods to run our SQL queries. But Go also provides *context-aware* variants of these two methods: `ExecContext()` and `QueryRowContext()`. These variants accept a `context.Context` instance as the first parameter which you can leverage to *terminate running database queries*.

This feature can be useful when *you have a SQL query that is taking longer to run than expected*. When this happens, it suggests a problem — either with that particular query or your database or application more generally — and you probably want to cancel the query (in order to free up resources), log an error for further investigation, and return a `500 Internal Server Error` response to the client.

In this chapter we'll update our application to do exactly that.

Mimicking a long-running query

To help demonstrate how this all works, let's start by adapting our database model's `Get()` method so that it mimics a long-running query. Specifically, we'll update our SQL query to return a `pg_sleep(10)` value, which will make PostgreSQL sleep for 10 seconds before returning its result.

File: internal/data/movies.go

```
package data

...

func (m MovieModel) Get(id int64) (*Movie, error) {
    if id < 1 {
        return nil, ErrRecordNotFound
    }

    // Update the query to return pg_sleep(10) as the first value.
    query := `
        SELECT pg_sleep(10), id, created_at, title, year, runtime, genres, version
        FROM movies
        WHERE id = $1`

    var movie Movie

    // Importantly, update the Scan() parameters so that the pg_sleep(10) return value
    // is scanned into a []byte slice.
    err := m.DB.QueryRow(query, id).Scan(
        &[]byte{}, // Add this line.
        &movie.ID,
        &movie.CreatedAt,
        &movie.Title,
        &movie.Year,
        &movie.Runtime,
        pq.Array(&movie.Genres),
        &movie.Version,
    )

    if err != nil {
        switch {
        case errors.Is(err, sql.ErrNoRows):
            return nil, ErrRecordNotFound
        default:
            return nil, err
        }
    }

    return &movie, nil
}

...
```

If you restart the application and make a request to the **GET /v1/movies/:id** endpoint, you should find that the request hangs for 10 seconds before you finally get a successful response displaying the movie information. Similar to this:

```
$ curl -w '\nTime: %{time_total}s \n' localhost:4000/v1/movies/1
{
  "movie": {
    "id": 1,
    "title": "Moana",
    "year": 2015,
    "runtime": "107 mins",
    "genres": [
      "animation",
      "adventure"
    ],
    "version": 1
  }
}

Time: 10.013534s
```

Note: In the `curl` command above we're using the `-w` flag to annotate the HTTP response with the total time taken for the command to complete. For more details about the timing information available in curl please see [this excellent blog post](#).

Adding a query timeout

Now that we've got some code that mimics a long-running query, let's enforce a timeout so that the SQL query is automatically canceled if it doesn't complete within 3 seconds.

To do this we need to:

1. Use the `context.WithTimeout()` function to create a `context.Context` instance with a 3-second timeout deadline.
2. Execute the SQL query using the `QueryRowContext()` method, passing the `context.Context` instance as a parameter.

I'll demonstrate:

File: internal/data/movies.go

```
package data

import (
    "context" // New import
    "database/sql"
    "errors"
    "time"

    "greenlight.alexedwards.net/internal/validator"

    "github.com/lib/pq"
)

...

func (m MovieModel) Get(id int64) (*Movie, error) {
    if id < 1 {
        return nil, ErrRecordNotFound
    }

    query := `
        SELECT pg_sleep(10), id, created_at, title, year, runtime, genres, version
        FROM movies
        WHERE id = $1`

    var movie Movie

    // Use the context.WithTimeout() function to create a context.Context which carries a
    // 3-second timeout deadline. Note that we're using the empty context.Background()
    // as the 'parent' context.
    ctx, cancel := context.WithTimeout(context.Background(), 3*time.Second)

    // Importantly, use defer to make sure that we cancel the context before the Get()
    // method returns.
    defer cancel()

    // Use the QueryRowContext() method to execute the query, passing in the context
    // with the deadline as the first argument.
    err := m.DB.QueryRowContext(ctx, query, id).Scan(
        &[]byte{},
        &movie.ID,
        &movie.CreatedAt,
        &movie.Title,
        &movie.Year,
        &movie.Runtime,
        pq.Array(&movie.Genres),
        &movie.Version,
    )

    if err != nil {
        switch {
            case errors.Is(err, sql.ErrNoRows):
                return nil, ErrRecordNotFound
            default:
                return nil, err
        }
    }

    return &movie, nil
}

...
```

There are a couple of things in the code above that I'd like to emphasize and explain:

- The `defer cancel()` line is necessary because it ensures that the resources associated with our context will always be released before the `Get()` method returns, thereby preventing a memory leak. Without it, the resources won't be released until either the 3-second timeout is hit or the parent context (which in this specific example is `context.Background()`) is canceled.
- The timeout countdown begins from the moment that the context is created using `context.WithTimeout()`. Any time spent executing code between creating the context and calling `QueryRowContext()` will count towards the timeout.

OK, let's try this out.

If you restart the application and make another request to the `GET /v1/movies/:id` endpoint, you should now get an error response similar to this after a 3-second delay:

```
$ curl -w '\nTime: %{time_total}s \n' localhost:4000/v1/movies/1
{
  "error": "the server encountered a problem and could not process your request"
}

Time: 3.025179s
```

If you go back to the terminal window which is running the application, you should also see a log line with the error message `"pq: canceling statement due to user request"`. Like so:

```
$ go run ./cmd/api
2021/04/08 14:14:52 database connection pool established
2021/04/08 14:14:52 starting development server on :4000
2021/04/08 14:14:57 pq: canceling statement due to user request
```

At first the wording of this error message might seem odd... until you learn that the message “canceling statement due to user request” is coming from PostgreSQL. In that light it makes sense: *our application* is the user, and we’re purposefully canceling the query after 3 seconds.

So, this is actually really good, and things are working as we would hope.

After 3 seconds, the context timeout is reached and our `pq` database driver sends a cancellation signal to PostgreSQL[†]. PostgreSQL then terminates the running query, the corresponding resources are freed-up, and it returns the error message that we see above. The client is then sent a `500 Internal Server Error` response, and the error is logged so

that we know something has gone wrong.

† More precisely, our context (the one with the 3-second timeout) has a **Done** channel, and when the timeout is reached the **Done** channel will be closed. While the SQL query is running, our database driver **pq** is also running a background goroutine which listens on this **Done** channel. If the channel gets closed, then **pq** sends a cancellation signal to PostgreSQL. PostgreSQL terminates the query, and then sends the error message that we see above as a response to the original **pq** goroutine. That error message is then returned to our database model's **Get()** method.

Timeouts outside of PostgreSQL

There's another important thing to point out here: *it's possible that the timeout deadline will be hit before the PostgreSQL query even starts.*

You might remember that earlier in the book we configured our **sql.DB** connection pool to allow a maximum of 25 open connections. If all those connections are in-use, then any additional queries will be 'queued' by **sql.DB** until a connection becomes available. In this scenario — or any other which causes a delay — it's possible that the timeout deadline will be hit before a free database connection even becomes available. If this happens then **QueryRowContext()** will return a **context.DeadlineExceeded** error.

In fact, we can demonstrate this in our application by setting the maximum open connections to 1 and making two concurrent requests to our endpoint. Go ahead and restart the API using a **-db-max-open-conns=1** flag, like so:

```
$ go run ./cmd/api -db-max-open-conns=1
2020/12/04 11:50:39 database connection pool established
2020/12/04 11:50:39 starting development server on :4000
```

Then in another terminal window make two requests to the **GET /v1/movies/:id** endpoint at the same time. At the moment that the 3-second timeout is reached, we should have one running SQL query and the other still 'queued' in the **sql.DB** connection pool. You should get two error responses which look like this:

```
$ curl localhost:4000/v1/movies/1 & curl localhost:4000/v1/movies/1 &
[1] 33221
[2] 33222
$ {
  "error": "the server encountered a problem and could not process your request"
}
{
  "error": "the server encountered a problem and could not process your request"
}
```

When you now head back to your original terminal, you should see two different error messages:

```
$ go run ./cmd/api -db-max-open-conns=1
2021/04/08 14:21:36 database connection pool established
2021/04/08 14:21:36 starting development server on :4000
2021/04/08 14:21:50 context deadline exceeded
2021/04/08 14:21:50 pq: canceling statement due to user request
```

Here the `pq: canceling statement due to user request` error message relates to the running SQL query being terminated, whereas the `context deadline exceeded` message relates to the queued SQL query being canceled *before a free database connection even became available*.

In a similar vein, it's also possible that the timeout deadline will be hit later on when the data returned from the query is being processed with `Scan()`. If this happens, then `Scan()` will also return a `context.DeadlineExceeded` error.

Updating our database model

Let's quickly update our database model to use a 3-second timeout deadline for all our operations. While we're at it, we'll remove the `pg_sleep(10)` clause from our `Get()` method too.

```
File: internal/data/movies.go

package data

...

func (m MovieModel) Insert(movie *Movie) error {
    query := `
        INSERT INTO movies (title, year, runtime, genres)
        VALUES ($1, $2, $3, $4)
        RETURNING id, created_at, version`

    args := []interface{}{movie.Title, movie.Year, movie.Runtime, pq.Array(movie.Genres)}

    // Create a context with a 3-second timeout.
```



```

    ctx, cancel := context.WithTimeout(context.Background(), 3*time.Second)
    defer cancel()

    // Use QueryRowContext() and pass the context as the first argument.
    return m.DB.QueryRowContext(ctx, query, args...).Scan(&movie.ID, &movie.CreatedAt, &movie.Version)
}

func (m MovieModel) Get(id int64) (*Movie, error) {
    if id < 1 {
        return nil, ErrRecordNotFound
    }

    // Remove the pg_sleep(10) clause.
    query := `
        SELECT id, created_at, title, year, runtime, genres, version
        FROM movies
        WHERE id = $1`

    var movie Movie

    ctx, cancel := context.WithTimeout(context.Background(), 3*time.Second)
    defer cancel()

    // Remove &[]byte{} from the first Scan() destination.
    err := m.DB.QueryRowContext(ctx, query, id).Scan(
        &movie.ID,
        &movie.CreatedAt,
        &movie.Title,
        &movie.Year,
        &movie.Runtime,
        pq.Array(&movie.Genres),
        &movie.Version,
    )

    if err != nil {
        switch {
        case errors.Is(err, sql.ErrNoRows):
            return nil, ErrRecordNotFound
        default:
            return nil, err
        }
    }

    return &movie, nil
}

func (m MovieModel) Update(movie *Movie) error {
    query := `
        UPDATE movies
        SET title = $1, year = $2, runtime = $3, genres = $4, version = version + 1
        WHERE id = $5 AND version = $6
        RETURNING version`

    args := []interface{}{
        movie.Title,
        movie.Year,
        movie.Runtime,
        pq.Array(movie.Genres),
        movie.ID,
        movie.Version,
    }

    // Create a context with a 3-second timeout.
    ctx, cancel := context.WithTimeout(context.Background(), 3*time.Second)
    defer cancel()

```

```

// Use QueryRowContext() and pass the context as the first argument.
err := m.DB.QueryRowContext(ctx, query, args...).Scan(&movie.Version)
if err != nil {
    switch {
    case errors.Is(err, sql.ErrNoRows):
        return ErrEditConflict
    default:
        return err
    }
}

return nil
}

func (m MovieModel) Delete(id int64) error {
    if id < 1 {
        return ErrRecordNotFound
    }

    query := `
        DELETE FROM movies
        WHERE id = $1`

    // Create a context with a 3-second timeout.
    ctx, cancel := context.WithTimeout(context.Background(), 3*time.Second)
    defer cancel()

    // Use ExecContext() and pass the context as the first argument.
    result, err := m.DB.ExecContext(ctx, query, id)
    if err != nil {
        return err
    }

    rowsAffected, err := result.RowsAffected()
    if err != nil {
        return err
    }

    if rowsAffected == 0 {
        return ErrRecordNotFound
    }

    return nil
}

```

Additional Information

Using the request context

As an alternative to the pattern that we’ve used in the code above, we could create a context with a timeout *in our handlers* using the *request context* as the parent — and then pass that on to our database model.

But — and it’s a big but — doing this introduces a lot of behavioral complexity, and for *most* applications the benefits aren’t large enough to make the trade-off worthwhile.

The details behind this are very interesting, but also quite intricate and heavy-going. For that reason I've discussed it further in [this appendix](#).

Filtering, Sorting, and Pagination

In this section of the book we're going to focus on building up the functionality for a new `GET /v1/movies` endpoint, which will return the details of multiple movies in a JSON array.

Method	URL Pattern	Handler	Action
GET	/v1/healthcheck	healthcheckHandler	Show application information
GET	/v1/movies	listMoviesHandler	Show the details of all movies
POST	/v1/movies	createMovieHandler	Create a new movie
GET	/v1/movies/:id	showMovieHandler	Show the details of a specific movie
PATCH	/v1/movies/:id	updateMovieHandler	Update the details of a specific movie
DELETE	/v1/movies/:id	deleteMovieHandler	Delete a specific movie

We'll develop the functionality for this endpoint incrementally, starting out by returning data for *all* movies and then gradually making it more useful and usable by adding filtering, sorting, and pagination functionality.

In this section you'll learn how to:

- Return the details of multiple resources in a single JSON response.
- Accept and apply optional filter parameters to narrow down the returned data set.
- Implement full-text search on your database fields using PostgreSQL's inbuilt functionality.
- Accept and safely apply sort parameters to change the order of results in the data set.
- Develop a pragmatic, reusable, pattern to support pagination on large data sets, and return pagination metadata in your JSON responses.

Parsing Query String Parameters

Over the next few chapters, we're going to configure the `GET /v1/movies` endpoint so that a client can control which movie records are returned via *query string parameters*. For example:

```
/v1/movies?title=godfather&genres=crime,drama&page=1&page_size=5&sort=-year
```

If a client sends a query string like this, it is essentially saying to our API: “*please return the first 5 records where the movie name includes `godfather` and the genres include `crime` and `drama`, sorted by descending release year*”.

Note: In the `sort` parameter we will use the `-` character to denote descending sort order. So, for example, the parameter `sort=title` implies an ascending alphabetical sort on movie title, whereas `sort=-title` implies a descending sort.

So the first thing we're going to look at is *how to parse these query string parameters in our Go code*.

As you can hopefully remember from *Let's Go*, we can retrieve the query string data from a request by calling the `r.URL.Query()` method. This returns a `url.Values` type, which is basically a map holding the query string data.

We can then extract values from this map using the `Get()` method, which will return the value for a specific key as a `string` type, or the empty string `""` if no matching key exists in the query string.

In our case, we'll need to carry out extra post-processing on some of these query string values too. Specifically:

- The `genres` parameter will potentially contain multiple *comma-separated values* — like `genres=crime,drama`. We will want to split these values apart and store them in a `[]string` slice.
- The `page` and `page_size` parameters will contain numbers, and we will want to convert these query string values into Go `int` types.

In addition to that:

- There are some validation checks that we'll want to apply to the query string values, like making sure that `page` and `page_size` are not negative numbers.
- We want our application to set some sensible *default values* in case parameters like `page`, `page_size` and `sort` aren't provided by the client.

Creating helper functions

To assist with this, we're going to create three new helper functions: `readString()`, `readInt()` and `readCSV()`. We'll use these helpers to extract and parse values from the query string, or return a default 'fallback' value if necessary.

Head to your `cmd/api/helpers.go` file and add the following code:

File: `cmd/api/helpers.go`

```
package main

import (
    "encoding/json"
    "errors"
    "fmt"
    "io"
    "net/http"
    "net/url" // New import
    "strconv"
    "strings"

    "greenlight.alexedwards.net/internal/validator" // New import
    "github.com/julienschmidt/httprouter"
)

...

// The readString() helper returns a string value from the query string, or the provided
// default value if no matching key could be found.
func (app *application) readString(qs url.Values, key string, defaultValue string) string {
    // Extract the value for a given key from the query string. If no key exists this
    // will return the empty string "".
    s := qs.Get(key)

    // If no key exists (or the value is empty) then return the default value.
    if s == "" {
        return defaultValue
    }

    // Otherwise return the string.
    return s
}

// The readCSV() helper reads a string value from the query string and then splits it
// into a slice on the comma character. If no matching key could be found, it returns
// the provided default value.
func (app *application) readCSV(qs url.Values, key string, defaultValue []string) []string {
    // Extract the value from the query string.
    csv := qs.Get(key)
```

```

// If no key exists (or the value is empty) then return the default value.
if csv == "" {
    return defaultValue
}

// Otherwise parse the value into a []string slice and return it.
return strings.Split(csv, ",")
}

// The readInt() helper reads a string value from the query string and converts it to an
// integer before returning. If no matching key could be found it returns the provided
// default value. If the value couldn't be converted to an integer, then we record an
// error message in the provided Validator instance.
func (app *application) readInt(qs url.Values, key string, defaultValue int, v *validator.Validator) int {
    // Extract the value from the query string.
    s := qs.Get(key)

    // If no key exists (or the value is empty) then return the default value.
    if s == "" {
        return defaultValue
    }

    // Try to convert the value to an int. If this fails, add an error message to the
    // validator instance and return the default value.
    i, err := strconv.Atoi(s)
    if err != nil {
        v.AddError(key, "must be an integer value")
        return defaultValue
    }

    // Otherwise, return the converted integer value.
    return i
}

```

Adding the API handler and route

Next up, let's create a new `listMoviesHandler` for our `GET /v1/movies` endpoint. For now, this handler will simply parse the request query string using the helpers we just made, and then dump the contents out in a HTTP response.

If you're following along, go ahead and create the `listMoviesHandler` like so:

File: cmd/api/movies.go

```
package main

...

func (app *application) listMoviesHandler(w http.ResponseWriter, r *http.Request) {
    // To keep things consistent with our other handlers, we'll define an input struct
    // to hold the expected values from the request query string.
    var input struct {
        Title    string
        Genres    []string
        Page     int
        PageSize int
        Sort     string
    }

    // Initialize a new Validator instance.
    v := validator.New()

    // Call r.URL.Query() to get the url.Values map containing the query string data.
    qs := r.URL.Query()

    // Use our helpers to extract the title and genres query string values, falling back
    // to defaults of an empty string and an empty slice respectively if they are not
    // provided by the client.
    input.Title = app.readString(qs, "title", "")
    input.Genres = app.readCSV(qs, "genres", []string{})

    // Get the page and page_size query string values as integers. Notice that we set
    // the default page value to 1 and default page_size to 20, and that we pass the
    // validator instance as the final argument here.
    input.Page = app.readInt(qs, "page", 1, v)
    input.PageSize = app.readInt(qs, "page_size", 20, v)

    // Extract the sort query string value, falling back to "id" if it is not provided
    // by the client (which will imply a ascending sort on movie ID).
    input.Sort = app.readString(qs, "sort", "id")

    // Check the Validator instance for any errors and use the failedValidationResponse()
    // helper to send the client a response if necessary.
    if !v.Valid() {
        app.failedValidationResponse(w, r, v.Errors)
        return
    }

    // Dump the contents of the input struct in a HTTP response.
    fmt.Fprintf(w, "%+v\n", input)
}
```

Then we need to create the `GET /v1/movies` route in our `cmd/api/routes.go` file, like so:

File: cmd/api/routes.go

```
package main

...

func (app *application) routes() *httprouter.Router {
    router := httprouter.New()

    router.NotFound = http.HandlerFunc(app.notFoundResponse)
    router.MethodNotAllowed = http.HandlerFunc(app.methodNotAllowedResponse)

    router.HandleFunc(http.MethodGet, "/v1/healthcheck", app.healthcheckHandler)

    // Add the route for the GET /v1/movies endpoint.
    router.HandleFunc(http.MethodGet, "/v1/movies", app.listMoviesHandler)
    router.HandleFunc(http.MethodPost, "/v1/movies", app.createMovieHandler)
    router.HandleFunc(http.MethodGet, "/v1/movies/:id", app.showMovieHandler)
    router.HandleFunc(http.MethodPatch, "/v1/movies/:id", app.updateMovieHandler)
    router.HandleFunc(http.MethodDelete, "/v1/movies/:id", app.deleteMovieHandler)

    return router
}
```

And with that, now we're ready to see this in action!

Go ahead and try sending a request to the **GET /v1/movies** endpoint containing the expected query string parameters, like below.

Important: When using curl to send a request containing more than one query string parameter, you **must** wrap the URL in quotes for it to work correctly.

```
$ curl "localhost:4000/v1/movies?title=godfather&genres=crime,drama&page=1&page_size=5&sort=year"
{Title:godfather Genres:[crime drama] Page:1 PageSize:5 Sort:year}
```

That's looking good — we can see that the values provided in our query string have all been parsed correctly and are included in the **input** struct.

If you want, you can also try making a request with *no* query string parameters. In this case, you should see that the values in the **input** struct take on the defaults we specified in our **listMoviesHandler** code. Like so:

```
$ curl localhost:4000/v1/movies
{Title: Genres:[] Page:1 PageSize:20 Sort:id}
```

Creating a Filters struct

The `page`, `page_size` and `sort` query string parameters are things that you'll potentially want to use on other endpoints in your API too. So, to help make this easier, let's quickly split them out into a reusable `Filters` struct.

If you're following along, go ahead and create a new `internal/data/filters.go` file:

```
$ touch internal/data/filters.go
```

And then add the following code:

```
File: internal/data/filters.go
```

```
package data

type Filters struct {
    Page      int
    PageSize  int
    Sort      string
}
```

Once that's done, head back to your `listMoviesHandler` and update it to use the new `Filters` struct like so:

File: cmd/api/movies.go

```
package main

...

func (app *application) listMoviesHandler(w http.ResponseWriter, r *http.Request) {
    // Embed the new Filters struct.
    var input struct {
        Title  string
        Genres []string
        data.Filters
    }

    v := validator.New()

    qs := r.URL.Query()

    input.Title = app.readString(qs, "title", "")
    input.Genres = app.readCSV(qs, "genres", []string{})

    // Read the page and page_size query string values into the embedded struct.
    input.Filters.Page = app.readInt(qs, "page", 1, v)
    input.Filters.PageSize = app.readInt(qs, "page_size", 20, v)

    // Read the sort query string value into the embedded struct.
    input.Filters.Sort = app.readString(qs, "sort", "id")

    if !v.Valid() {
        app.failedValidationResponse(w, r, v.Errors)
        return
    }

    fmt.Fprintf(w, "%+v\n", input)
}
```

At this point, you should be able to run the API again and everything should continue to work like before.

Validating Query String Parameters

Thanks to the `readInt()` helper that we made in the previous chapter, our API should already be returning validation errors if the `page` and `page_size` query string parameters don't contain integer values. Go ahead and try it out, if you like:

```
$ curl "localhost:4000/v1/movies?page=abc&page_size=abc"
{
  "error": {
    "page": "must be an integer value",
    "page_size": "must be an integer value"
  }
}
```

But we still need to perform some additional sanity checks on the query string values provided by the client. In particular, we want to check that:

- The `page` value is between 1 and 10,000,000.
- The `page_size` value is between 1 and 100.
- The `sort` parameter contains a known and supported value for our movies table. Specifically, we'll allow `"id"`, `"title"`, `"year"`, `"runtime"`, `"-id"`, `"-title"`, `"-year"` or `"-runtime"`.

To fix this, let's open up the `internal/data/filters.go` file and create a new `ValidateFilters()` function which conducts these checks on the values.

We'll follow the same pattern that we used for the `ValidateMovie()` function earlier on to do this, like so:

File: internal/data/filters.go

```
package data

import (
    "greenlight.alexedwards.net/internal/validator" // New import
)

// Add a SortSafelist field to hold the supported sort values.
type Filters struct {
    Page          int
    PageSize      int
    Sort          string
    SortSafelist []string
}

func ValidateFilters(v *validator.Validator, f Filters) {
    // Check that the page and page_size parameters contain sensible values.
    v.Check(f.Page > 0, "page", "must be greater than zero")
    v.Check(f.Page <= 10_000_000, "page", "must be a maximum of 10 million")
    v.Check(f.PageSize > 0, "page_size", "must be greater than zero")
    v.Check(f.PageSize <= 100, "page_size", "must be a maximum of 100")

    // Check that the sort parameter matches a value in the safelist.
    v.Check(validator.In(f.Sort, f.SortSafelist...), "sort", "invalid sort value")
}
```

Then we need to update our `listMoviesHandler` to set the supported values in the `SortSafelist` field, and subsequently call this new `ValidateFilters()` function.

File: cmd/api/movies.go

```
package main

...

func (app *application) listMoviesHandler(w http.ResponseWriter, r *http.Request) {
    var input struct {
        Title string
        Genres []string
        data.Filters
    }

    v := validator.New()

    qs := r.URL.Query()

    input.Title = app.readString(qs, "title", "")
    input.Genres = app.readCSV(qs, "genres", []string{})

    input.Filters.Page = app.readInt(qs, "page", 1, v)
    input.Filters.PageSize = app.readInt(qs, "page_size", 20, v)

    input.Filters.Sort = app.readString(qs, "sort", "id")
    // Add the supported sort values for this endpoint to the sort safelist.
    input.Filters.SortSafelist = []string{"id", "title", "year", "runtime", "-id", "-title", "-year", "-runtime"}

    // Execute the validation checks on the Filters struct and send a response
    // containing the errors if necessary.
    if data.ValidateFilters(v, input.Filters); !v.Valid() {
        app.failedValidationResponse(w, r, v.Errors)
        return
    }

    fmt.Fprintf(w, "%+v\n", input)
}
```

If you restart the API and try making a request with some invalid `page`, `page_size` and `sort` parameters, you should now receive an error response containing the relevant validation failure messages. Similar to this:

```
$ curl "localhost:4000/v1/movies?page=-1&page_size=-1&sort=foo"
{
  "error": {
    "page": "must be greater than zero",
    "page_size": "must be greater than zero",
    "sort": "invalid sort value"
  }
}
```

Feel free to test this out more if you like, and try sending different query string values until you're confident that the validation checks are all working correctly.

Listing Data

OK, let's move on and get our `GET /v1/movies` endpoint returning some real data.

For now, we'll ignore any query string values provided by the client and return *all* movie records, sorted by movie ID. This will give us a solid base from which we can develop the more specialized functionality around filtering, sorting, and pagination.

Our aim in this chapter will be to get the endpoint to return a JSON response containing an array of all movies, similar to this:

```
{
  "movies": [
    {
      "id": 1,
      "title": "Moana",
      "year": 2015,
      "runtime": "107 mins",
      "genres": [
        "animation",
        "adventure"
      ],
      "version": 1
    },
    {
      "id": 2,
      "title": "Black Panther",
      "year": 2018,
      "runtime": "134 mins",
      "genres": [
        "sci-fi",
        "action",
        "adventure"
      ],
      "version": 2
    },
    ... etc.
  ]
}
```

Updating the application

To retrieve this data from our PostgreSQL database, let's create a new `GetAll()` method on our database model which executes the following SQL query:

```
SELECT id, created_at, title, year, runtime, genres, version
FROM movies
ORDER BY id
```

Because we're expecting this SQL query to return multiple records, we'll need to run it using Go's `QueryContext()` method. We already explained how this works in detail in *Let's Go*, so let's jump into the code:

File: internal/data/movies.go

```
package data

...

// Create a new GetAll() method which returns a slice of movies. Although we're not
// using them right now, we've set this up to accept the various filter parameters as
// arguments.
func (m MovieModel) GetAll(title string, genres []string, filters Filters) ([]*Movie, error) {
    // Construct the SQL query to retrieve all movie records.
    query := `
        SELECT id, created_at, title, year, runtime, genres, version
        FROM movies
        ORDER BY id`

    // Create a context with a 3-second timeout.
    ctx, cancel := context.WithTimeout(context.Background(), 3*time.Second)
    defer cancel()

    // Use QueryContext() to execute the query. This returns a sql.Rows resultset
    // containing the result.
    rows, err := m.DB.QueryContext(ctx, query)
    if err != nil {
        return nil, err
    }

    // Importantly, defer a call to rows.Close() to ensure that the resultset is closed
    // before GetAll() returns.
    defer rows.Close()

    // Initialize an empty slice to hold the movie data.
    movies := []*Movie{}

    // Use rows.Next() to iterate through the rows in the resultset.
    for rows.Next() {
        // Initialize an empty Movie struct to hold the data for an individual movie.
        var movie Movie

        // Scan the values from the row into the Movie struct. Again, note that we're
        // using the pq.Array() adapter on the genres field here.
        err := rows.Scan(
            &movie.ID,
            &movie.CreatedAt,
            &movie.Title,
            &movie.Year,
            &movie.Runtime,
            pq.Array(&movie.Genres),
            &movie.Version,
        )
        if err != nil {
            return nil, err
        }
    }
}
```



```

    }

    // Add the Movie struct to the slice.
    movies = append(movies, &movie)
}

// When the rows.Next() loop has finished, call rows.Err() to retrieve any error
// that was encountered during the iteration.
if err = rows.Err(); err != nil {
    return nil, err
}

// If everything went OK, then return the slice of movies.
return movies, nil
}

```

Next up, we need to adapt the `listMoviesHandler` so that it calls the new `GetAll()` method to retrieve the movie data, and then writes this data as a JSON response.

If you're following along, go ahead and update the handler like so:

File: cmd/api/movies.go

```
package main

...

func (app *application) listMoviesHandler(w http.ResponseWriter, r *http.Request) {
    var input struct {
        Title string
        Genres []string
        data.Filters
    }

    v := validator.New()

    qs := r.URL.Query()

    input.Title = app.readString(qs, "title", "")
    input.Genres = app.readCSV(qs, "genres", []string{})

    input.Filters.Page = app.readInt(qs, "page", 1, v)
    input.Filters.PageSize = app.readInt(qs, "page_size", 20, v)

    input.Filters.Sort = app.readString(qs, "sort", "id")
    input.Filters.SortSafelist = []string{"id", "title", "year", "runtime", "-id", "-title", "-year", "-runtime"}

    if data.ValidateFilters(v, input.Filters); !v.Valid() {
        app.failedValidationResponse(w, r, v.Errors)
        return
    }

    // Call the GetAll() method to retrieve the movies, passing in the various filter
    // parameters.
    movies, err := app.models.Movies.GetAll(input.Title, input.Genres, input.Filters)
    if err != nil {
        app.serverErrorResponse(w, r, err)
        return
    }

    // Send a JSON response containing the movie data.
    err = app.writeJSON(w, http.StatusOK, envelope{"movies": movies}, nil)
    if err != nil {
        app.serverErrorResponse(w, r, err)
    }
}
```

And now we should be ready to try this out.

Go ahead and restart the API, then when you make a **GET /v1/movies** request you should see the slice of movies returned by **GetAll()** rendered as a JSON array. Similar to this:

```
$ curl localhost:4000/v1/movies
```

```
{
  "movies": [
    {
      "id": 1,
      "title": "Moana",
      "year": 2016,
      "runtime": "107 mins",
      "genres": [
        "animation",
        "adventure"
      ],
      "version": 1
    },
    {
      "id": 2,
      "title": "Black Panther",
      "year": 2018,
      "runtime": "134 mins",
      "genres": [
        "sci-fi",
        "action",
        "adventure"
      ],
      "version": 2
    },
    {
      "id": 4,
      "title": "The Breakfast Club",
      "year": 1985,
      "runtime": "97 mins",
      "genres": [
        "comedy"
      ],
      "version": 5
    }
  ]
}
```

Filtering Lists

In this chapter we're going to start putting our query string parameters to use, so that clients can search for movies with a specific title or genres.

Specifically, we'll build a [reductive filter](#) which allows clients to search based on a case-insensitive exact match for movie title and/or one or more movie genres. For example:

```
// List all movies.  
/v1/movies  
  
// List movies where the title is a case-insensitive exact match for 'black panther'.  
/v1/movies?title=black+panther  
  
// List movies where the genres includes 'adventure'.  
/v1/movies?genres=adventure  
  
// List movies where the title is a case-insensitive exact match for 'moana' AND the  
// genres include both 'animation' AND 'adventure'.  
/v1/movies?title=moana&genres=animation,adventure
```

Note: The **+** symbol in the query strings above is a URL-encoded space character. Alternatively you could use **%20** instead... either will work in the context of a query string.

Dynamic filtering in the SQL query

The hardest part of building a dynamic filtering feature like this is the SQL query to retrieve the data — we need it to work with no filters, filters on both **title** and **genres**, or a filter on only one of them.

To deal with this, one option is to build up the SQL query dynamically at runtime... with the necessary SQL for each filter concatenated or interpolated into the **WHERE** clause. But this approach can make your code messy and difficult to understand, especially for large queries which need to support lots of filter options.

In this book we'll opt for a different technique and use a fixed SQL query which looks like this:

```
SELECT id, created_at, title, year, runtime, genres, version
FROM movies
WHERE (LOWER(title) = LOWER($1) OR $1 = '')
AND (genres @> $2 OR $2 = '{}')
ORDER BY id
```

This SQL query is designed so that each of the filters behaves like it is ‘optional’. For example, the condition `(LOWER(title) = LOWER($1) OR $1 = '')` will evaluate as `true` if the placeholder parameter `$1` is a case-insensitive match for the movie title *or* the placeholder parameter equals `''`. So this filter condition will essentially be ‘skipped’ when movie title being searched for is the empty string `''`.

The `(genres @> $2 OR $2 = '{}')` condition works in the same way. The `@>` symbol is the ‘contains’ operator for PostgreSQL arrays, and this condition will return `true` if each value in the placeholder parameter `$2` appears in the database `genres` field *or* the placeholder parameter contains an empty array.

You’ll remember that earlier in the book we set up our `listMoviesHandler` so that the empty string `''` and an empty slice are used as the default values for the `title` and `genres` filter parameters:

```
input.Title = app.readString(qs, "title", "")
input.Genres = app.readCSV(qs, "genres", []string{})
```

So, putting this all together, it means that if a client doesn’t provide a `title` parameter in their query string, then value for the `$1` placeholder will be the empty string `''`, and the filter condition in the SQL query will evaluate to `true` and act like it has been ‘skipped’. Likewise with the `genres` parameter.

Note: PostgreSQL also provides a range of other useful array operators and functions, including the `&&` ‘overlap’ operator, the `<@` ‘contained by’ operator, and the `array_length()` function. A complete list [can be found here](#).

Alright, let’s head back to our `internal/data/movies.go` file, and update the `GetAll()` method to use this new query. Like so:

File: internal/data/movies.go

```
package data

...

func (m MovieModel) GetAll(title string, genres []string, filters Filters) ([]*Movie, error) {
    // Update the SQL query to include the filter conditions.
    query := `
        SELECT id, created_at, title, year, runtime, genres, version
        FROM movies
        WHERE (LOWER(title) = LOWER($1) OR $1 = '')
        AND (genres @> $2 OR $2 = '{}')
        ORDER BY id`

    ctx, cancel := context.WithTimeout(context.Background(), 3*time.Second)
    defer cancel()

    // Pass the title and genres as the placeholder parameter values.
    rows, err := m.DB.QueryContext(ctx, query, title, pq.Array(genres))
    if err != nil {
        return nil, err
    }

    defer rows.Close()

    movies := []*Movie{}

    for rows.Next() {
        var movie Movie

        err := rows.Scan(
            &movie.ID,
            &movie.CreatedAt,
            &movie.Title,
            &movie.Year,
            &movie.Runtime,
            pq.Array(&movie.Genres),
            &movie.Version,
        )
        if err != nil {
            return nil, err
        }

        movies = append(movies, &movie)
    }

    if err = rows.Err(); err != nil {
        return nil, err
    }

    return movies, nil
}
```

Now let's restart the application and try this out, using the examples that we gave at the start of the chapter. If you've been following along, the responses should look similar to this:

```
$ curl "localhost:4000/v1/movies?title=black+panther"
{
  "movies": [
    {
      "id": 2,
      "title": "Black Panther",
      "year": 2018,
      "runtime": "134 mins",
      "genres": [
        "sci-fi",
        "action",
        "adventure"
      ],
      "version": 2
    }
  ]
}
```

```
$ curl "localhost:4000/v1/movies?genres=adventure"
{
  "movies": [
    {
      "id": 1,
      "title": "Moana",
      "year": 2015,
      "runtime": "107 mins",
      "genres": [
        "animation",
        "adventure"
      ],
      "version": 1
    },
    {
      "id": 2,
      "title": "Black Panther",
      "year": 2018,
      "runtime": "134 mins",
      "genres": [
        "sci-fi",
        "action",
        "adventure"
      ],
      "version": 2
    }
  ]
}
```

```
$ curl "localhost:4000/v1/movies?title=moana&genres=animation,adventure"
{
  "movies": [
    {
      "id": 1,
      "title": "Moana",
      "year": 2016,
      "runtime": "107 mins",
      "genres": [
        "animation",
        "adventure"
      ],
      "version": 1
    }
  ]
}
```

You can also try making a request with a filter that doesn't match any records. In this case, you should get an empty JSON array in the response like so:

```
$ curl "localhost:4000/v1/movies?genres=western"
{
  "movies": []
}
```

This is shaping up nicely. Our API endpoint is now returning the appropriately filtered movie records, and we have a pattern that we can easily extend to include other filtering rules in the future (such as a filter on the movie year or runtime) if we want to.

Full-Text Search

In this chapter we're going to make our movie title filter easier to use by adapting it to support *partial matches*, rather than requiring a match on the full title. So, for example, if a client wants to find *The Breakfast Club* they will be able to find it with just the query string `title=breakfast`.

There are a few different ways we could implement this feature in our codebase, but an effective and intuitive method (from a client point of view) is to leverage PostgreSQL's *full-text search* functionality, which allows you to perform 'natural language' searches on text fields in your database.

PostgreSQL full-text search is a powerful and highly-configurable tool, and explaining how it works and the available options in full could easily take up a whole book in itself. So we'll keep the explanations in this chapter high-level, and focus on the practical implementation.

To implement a basic full-text search on our `title` field, we're going to update our SQL query to look like this:

```
SELECT id, created_at, title, year, runtime, genres, version
FROM movies
WHERE (to_tsvector('simple', title) @@ plainto_tsquery('simple', $1) OR $1 = '')
AND (genres @> $2 OR $2 = '{}')
ORDER BY id
```

That looks pretty complicated at first glance, so let's break it down and explain what's going on.

The `to_tsvector('simple', title)` function takes a movie title and splits it into *lexemes*. We specify the `simple` configuration, which means that the lexemes are just lowercase versions of the words in the title[†]. For example, the movie title "`The Breakfast Club`" would be split into the lexemes `'breakfast'` `'club'` `'the'`.

[†] Other 'non-simple' configurations may apply additional rules to the lexemes, such as the removal of common words or applying language-specific stemming.

The `plainto_tsquery('simple', $1)` function takes a search value and turns it into a formatted *query term* that PostgreSQL full-text search can understand. It normalizes the

search value (again using the `simple` configuration), strips any special characters, and inserts the *and operator* `&` between the words. As an example, the search value `"The Club"` would result in the query term `'the' & 'club'`.

The `@@` operator is the matches operator. In our statement we are using it to check whether the generated *query term matches the lexemes*. To continue the example, the query term `'the' & 'club'` will match rows which contain *both* lexemes `'the'` and `'club'`.

There are a lot of specialist words in the paragraphs above, but if we illustrate it with a couple of examples it's actually very intuitive:

```
// Return all movies where the title includes the case-insensitive word 'panther'.
/v1/movies?title=panther

// Return all movies where the title includes the case-insensitive words 'the' and
// 'club'.
/v1/movies?title=the+club
```

Let's go ahead and put this into action. Open up your `internal/data/movies.go` file and update the `GetAll()` method to use the new SQL query like so:

```
File: internal/data/movies.go

package data

...

func (m MovieModel) GetAll(title string, genres []string, filters Filters) ([]*Movie, error) {
    // Use full-text search for the title filter.
    query := `
        SELECT id, created_at, title, year, runtime, genres, version
        FROM movies
        WHERE (to_tsvector('simple', title) @@ plainto_tsquery('simple', $1) OR $1 = '')
        AND (genres @> $2 OR $2 = '{}')
        ORDER BY id`

    // Nothing else below needs to change.
    ...
}
```

If you're following along, restart the application and try making some requests with different values for the movie title. You should find that partial searches now work as we described above.

For example:

```
$ curl "localhost:4000/v1/movies?title=panther"
{
  "movies": [
    {
      "id": 2,
      "title": "Black Panther",
      "year": 2018,
      "runtime": "134 mins",
      "genres": [
        "sci-fi",
        "action",
        "adventure"
      ],
      "version": 2
    }
  ]
}

$ curl "localhost:4000/v1/movies?title=the+club"
{
  "movies": [
    {
      "id": 4,
      "title": "The Breakfast Club",
      "year": 1985,
      "runtime": "97 mins",
      "genres": [
        "comedy"
      ],
      "version": 5
    }
  ]
}
```

Adding indexes

To keep our SQL query performing quickly as the dataset grows, it's sensible to use indexes to help avoid full table scans and avoid generating the lexemes for the `title` field every time the query is run.

Note: If you are not familiar with the concept of indexes in SQL databases, the official PostgreSQL documentation provides a [good introduction](#) and a summary of different [index types](#). I recommend reading these quickly to give you an overview before continuing.

In our case it makes sense to create [GIN indexes](#) on both the `genres` field and the lexemes generated by `to_tsvector()`, both which are used in the `WHERE` clause of our SQL query.

If you're following along, go ahead and create a new pair of migration files:

```
$ migrate create -seq -ext .sql -dir ./migrations add_movies_indexes
```

Then add the following statements to the ‘up’ and ‘down’ migration files to create and drop the necessary indexes:

```
File: migrations/000003_add_movies_indexes.up.sql
```

```
CREATE INDEX IF NOT EXISTS movies_title_idx ON movies USING GIN (to_tsvector('simple', title));  
CREATE INDEX IF NOT EXISTS movies_genres_idx ON movies USING GIN (genres);
```

```
File: migrations/000003_add_movies_indexes.down.sql
```

```
DROP INDEX IF EXISTS movies_title_idx;  
DROP INDEX IF EXISTS movies_genres_idx;
```

Once that’s done, you should be able to execute the ‘up’ migration to add the indexes to your database:

```
$ migrate -path ./migrations -database $GREENLIGHT_DB_DSN up  
3/u add_movies_indexes (38.638777ms)
```

Additional Information

Non-simple configuration and more information

As mentioned above, you can also use a language-specific configuration for full-text searches instead of the **simple** configuration that we’re currently using. When you create lexemes or query terms using a language-specific configuration, it will strip out common words for the language and perform word **stemming**.

So, for example, if you use the **english** configuration, then the lexemes generated for "One Flew Over the Cuckoo's Nest" would be 'cuckoo' 'flew' 'nest' 'one'. Or with the **spanish** configuration, the lexemes for "Los lunes al sol" would be 'lun' 'sol'.

You can retrieve a list of all available configurations by running the **\df** meta-command in PostgreSQL:

```
postgres=# \dF
          List of text search configurations
 Schema | Name | Description
-----+-----+-----
 pg_catalog | arabic | configuration for arabic language
 pg_catalog | danish | configuration for danish language
 pg_catalog | dutch | configuration for dutch language
 pg_catalog | english | configuration for english language
 ...
```

And if you wanted to use the **english** configuration to search our movies, you could update the SQL query like so:

```
SELECT id, created_at, title, year, runtime, genres, version
FROM movies
WHERE (to_tsvector('english', title) @@ plainto_tsquery('english', $1) OR $1 = '')
AND (genres @> $2 OR $2 = '{}')
ORDER BY id
```

If you'd like to learn more about PostgreSQL full-text search then reading [this blog post](#) is a good next step, and the [official documentation](#) is also excellent.

Using STRPOS and ILIKE

If you don't want to use full-text search for the partial movie title lookup, some alternatives are the PostgreSQL **STRPOS()** function and **ILIKE** operator.

The PostgreSQL **STRPOS()** function allows you to check for the existence of a substring in a particular database field. We could use it in our SQL query like this:

```
SELECT id, created_at, title, year, runtime, genres, version
FROM movies
WHERE (STRPOS(LOWER(title), LOWER($1)) > 0 OR $1 = '')
AND (genres @> $2 OR $2 = '{}')
ORDER BY id
```

From a client perspective, the downside of this is that it may return some unintuitive results. For example, searching for **title=the** would return both *The Breakfast Club* and *Black Panther* in our dataset.

From a server perspective it's also not ideal for large datasets. Because there's no effective way to index the **title** field to see if the **STRPOS()** condition is met, it means the query could potentially require a full-table scan each time it is run.

Another option is the **ILIKE** operator, which allows you to find rows which match a specific (case-insensitive) pattern. We could use it in our SQL query like so:

```
SELECT id, created_at, title, year, runtime, genres, version
FROM movies
WHERE (title ILIKE $1 OR $1 = '')
AND (genres @> $2 OR $2 = '{}')
ORDER BY id
```

This approach would be better from a server point of view because it's possible to create an index on the **title** field using the **pg_trgm** extension and a GIN index (for details please see [this post](#)).

From the client side it's arguably better that the **STRPOS()** approach too, as they can control the matching behavior by prefixing/suffixing the search term with a **%** wildcard character (which will need to be escaped to **%25** in the URL query string). For example, to search for movies with a title that starts with “the”, a client could send the query string parameter **title=the%25**.

Sorting Lists

Let's now update the logic for our `GET /v1/movies` endpoint so that the client can control how the movies are sorted in the JSON response.

As we briefly explained earlier, we want to let the client control the sort order via a query string parameter in the format `sort={-}{field_name}`, where the optional `-` character is used to indicate a descending sort order. For example:

```
// Sort the movies on the title field in ascending alphabetical order.  
/v1/movies?sort=title  
  
// Sort the movies on the year field in descending numerical order.  
/v1/movies?sort=-year
```

Behind the scenes we will want to translate this into an `ORDER BY` clause in our SQL query, so that a query string parameter like `sort=-year` would result in a SQL query like this:

```
SELECT id, created_at, title, year, runtime, genres, version  
FROM movies  
WHERE (STRPOS(LOWER(title), LOWER($1)) > 0 OR $1 = '')  
AND (genres @> $2 OR $2 = '{}')  
ORDER BY year DESC --<-- Order the result by descending year
```

The difficulty here is that the values for the `ORDER BY` clause will need to be generated at runtime based on the query string values from the client. Ideally we'd use placeholder parameters to insert these dynamic values into our query, but unfortunately it's *not possible to use placeholder parameters for column names or SQL keywords* (including `ASC` and `DESC`).

So instead, we'll need to interpolate these dynamic values into our query using `fmt.Sprintf()` — making sure that the values are checked against a strict safelist first to prevent a SQL injection attack.

When working with PostgreSQL, it's also important to be aware that the order of returned rows *is only guaranteed by the rules that your `ORDER BY` clause imposes*. From the [official documentation](#):

If sorting is not chosen, the rows will be returned in an unspecified order. The actual order in that case will depend on the scan and join plan types and the order on disk, but it must not be relied on. A particular output ordering can only be guaranteed if the sort step is

explicitly chosen.

That means if we don't include an **ORDER BY** clause, then PostgreSQL may return the movies in any order — *and the order may or may not change each time the query is run.*

Likewise, in our database multiple movies will have the same **year** value. If we order based on the **year** column, then the movies are guaranteed be ordered by year, but the movies *for* a particular year could appear in any order at any time.

This point is particularly important in the context of an endpoint which provides pagination. We need to make sure that the order of movies is perfectly consistent between requests to prevent items in the list 'jumping' between the pages.

Fortunately, guaranteeing the order is simple — we just need to ensure that the **ORDER BY** clause always includes a primary key column (or another column with a unique constraint on it). So, in our case, we can apply a secondary sort on the **id** column to ensure an always-consistent order. Like so:

```
SELECT id, created_at, title, year, runtime, genres, version
FROM movies
WHERE (STRPOS(LOWER(title), LOWER($1)) > 0 OR $1 = '')
AND (genres @> $2 OR $2 = '{}')
ORDER BY year DESC, id ASC
```

Implementing sorting

To get the dynamic sorting working, let's begin by updating our **Filters** struct to include some **sortColumn()** and **sortDirection()** helpers that transform a query string value (like **-year**) into values we can use in our SQL query.

File: internal/data/filters.go

```
package data

import (
    "strings" // New import

    "greenlight.alexedwards.net/internal/validator"
)

type Filters struct {
    Page      int
    PageSize  int
    Sort       string
    SortSafelist []string
}

// Check that the client-provided Sort field matches one of the entries in our safelist
// and if it does, extract the column name from the Sort field by stripping the leading
// hyphen character (if one exists).
func (f Filters) sortColumn() string {
    for _, safeValue := range f.SortSafelist {
        if f.Sort == safeValue {
            return strings.TrimPrefix(f.Sort, "-")
        }
    }

    panic("unsafe sort parameter: " + f.Sort)
}

// Return the sort direction ("ASC" or "DESC") depending on the prefix character of the
// Sort field.
func (f Filters) sortDirection() string {
    if strings.HasPrefix(f.Sort, "-") {
        return "DESC"
    }

    return "ASC"
}

...
```

Notice that the `sortColumn()` function is constructed in such a way that it will panic if the client-provided `Sort` value doesn't match one of the entries in our safelist. In theory this shouldn't happen — the `Sort` value should have already been checked by calling the `ValidateFilters()` function — but this is a sensible failsafe to help stop a SQL injection attack occurring.

Now let's update our `internal/data/movies.go` file to call those methods, and interpolate the return values into our SQL query's `ORDER BY` clause. Like so:

File: internal/data/movies.go

```
package data

import (
    "context"
    "database/sql"
    "errors"
    "fmt" // New import
    "time"

    "greenlight.alexedwards.net/internal/validator"

    "github.com/lib/pq"
)

...

func (m MovieModel) GetAll(title string, genres []string, filters Filters) ([]*Movie, error) {
    // Add an ORDER BY clause and interpolate the sort column and direction. Importantly
    // notice that we also include a secondary sort on the movie ID to ensure a
    // consistent ordering.
    query := fmt.Sprintf(`
        SELECT id, created_at, title, year, runtime, genres, version
        FROM movies
        WHERE (to_tsvector('simple', title) @@ plainto_tsquery('simple', $1) OR $1 = '')
        AND (genres @> $2 OR $2 = '{}')
        ORDER BY %s %s, id ASC`, filters.sortColumn(), filters.sortDirection())

    // Nothing else below needs to change.
    ...
}
```

And once that's done, we should be ready to try this out.

Restart the application then, as an example, go ahead and try making a request for the movies sorted by descending **title**. You should get a response which looks like this:

```
$ curl "localhost:4000/v1/movies?sort=-title"
{
  "movies": [
    {
      "id": 4,
      "title": "The Breakfast Club",
      "year": 1985,
      "runtime": "97 mins",
      "genres": [
        "comedy"
      ],
      "version": 5
    },
    {
      "id": 1,
      "title": "Moana",
      "year": 2016,
      "runtime": "107 mins",
      "genres": [
        "animation",
        "adventure"
      ],
      "version": 1
    },
    {
      "id": 2,
      "title": "Black Panther",
      "year": 2018,
      "runtime": "134 mins",
      "genres": [
        "sci-fi",
        "action",
        "adventure"
      ],
      "version": 2
    }
  ]
}
```

In contrast, using a sort parameter of descending **runtime** should yield a response in a totally different order. Similar to this:

```
$ curl "localhost:4000/v1/movies?sort=-runtime"
```

```
{
  "movies": [
    {
      "id": 2,
      "title": "Black Panther",
      "year": 2018,
      "runtime": "134 mins",
      "genres": [
        "sci-fi",
        "action",
        "adventure"
      ],
      "version": 2
    },
    {
      "id": 1,
      "title": "Moana",
      "year": 2016,
      "runtime": "107 mins",
      "genres": [
        "animation",
        "adventure"
      ],
      "version": 1
    },
    {
      "id": 4,
      "title": "The Breakfast Club",
      "year": 1985,
      "runtime": "97 mins",
      "genres": [
        "comedy"
      ],
      "version": 5
    }
  ]
}
```

Paginating Lists

If you have an endpoint which returns a list with hundreds or thousands of records, then for performance or usability reasons you might want to implement some form of *pagination* on the endpoint — so that it only returns a subset of the records in a single HTTP response.

To help demonstrate how to do this, in this chapter we're going to update the `GET /v1/movies` endpoint so that it supports the concept of 'pages' and a client can request a specific 'page' of the movies list by using our `page` and `page_size` query string parameters. For example:

```
// Return the 5 records on page 1 (records 1-5 in the dataset)
/v1/movies?page=1&page_size=5

// Return the next 5 records on page 2 (records 6-10 in the dataset)
/v1/movies?page=2&page_size=5

// Return the next 5 records on page 3 (records 11-15 in the dataset)
/v1/movies?page=3&page_size=5
```

Basically, changing the `page_size` parameter will alter the number of movies that are shown on each 'page', and increasing the `page` parameter by one will show you the next 'page' of movies in the list.

The LIMIT and OFFSET clauses

Behind the scenes, the simplest way to support this style of pagination is by adding `LIMIT` and `OFFSET` clauses to our SQL query.

The `LIMIT` clause allows you to set the maximum number of records that a SQL query should return, and `OFFSET` allows you to 'skip' a specific number of rows before starting to return records from the query.

Within our application, we'll just need to translate the `page` and `page_size` values provided by the client to the appropriate `LIMIT` and `OFFSET` values for our SQL query. The math is pretty straightforward:

```
LIMIT = page_size
OFFSET = (page - 1) * page_size
```

Or to give a concrete example, if a client makes the following request:

```
/v1/movies?page_size=5&page=3
```

We would need to ‘translate’ this into the following SQL query:

```
SELECT id, created_at, title, year, runtime, genres, version
FROM movies
WHERE (to_tsvector('simple', title) @@ plainto_tsquery('simple', $1) OR $1 = '')
AND (genres @> $2 OR $2 = '{}')
ORDER BY %s %s, id ASC
LIMIT 5 OFFSET 10
```

Let’s start by adding some helper methods to our `Filters` struct for calculating the appropriate `LIMIT` and `OFFSET` values.

If you’re following along, go ahead and update the `internal/data/filters.go` file like so:

```
File: internal/data/filters.go

package data

...

type Filters struct {
    Page         int
    PageSize     int
    Sort         string
    SortSafelist []string
}

...

func (f Filters) limit() int {
    return f.PageSize
}

func (f Filters) offset() int {
    return (f.Page - 1) * f.PageSize
}

...
```

Note: In the `offset()` method there is the theoretical risk of an `integer overflow` as we are multiplying two `int` values together. However, this is mitigated by the validation rules we created in our `ValidateFilters()` function, where we enforced maximum values of `page_size=100` and `page=10000000` (10 million). This means that the value returned by `offset()` should never come close to overflowing.

Updating the database model

As the final stage in this process, we need to update our database model's `GetAll()` method to add the appropriate `LIMIT` and `OFFSET` clauses to the SQL query.

File: internal/data/movies.go

```
package data

...

func (m MovieModel) GetAll(title string, genres []string, filters Filters) ([]*Movie, error) {
    // Update the SQL query to include the LIMIT and OFFSET clauses with placeholder
    // parameter values.
    query := fmt.Sprintf(`
        SELECT id, created_at, title, year, runtime, genres, version
        FROM movies
        WHERE (to_tsvector('simple', title) @@ plainto_tsquery('simple', $1) OR $1 = '')
        AND (genres @> $2 OR $2 = '{}')
        ORDER BY %s %s, id ASC
        LIMIT $3 OFFSET $4`, filters.sortColumn(), filters.sortDirection())

    ctx, cancel := context.WithTimeout(context.Background(), 3*time.Second)
    defer cancel()

    // As our SQL query now has quite a few placeholder parameters, let's collect the
    // values for the placeholders in a slice. Notice here how we call the limit() and
    // offset() methods on the Filters struct to get the appropriate values for the
    // LIMIT and OFFSET clauses.
    args := []interface{}{title, pq.Array(genres), filters.limit(), filters.offset()}

    // And then pass the args slice to QueryContext() as a variadic parameter.
    rows, err := m.DB.QueryContext(ctx, query, args...)
    if err != nil {
        return nil, err
    }

    // Nothing else below needs to change.
    ...
}
```

Once that's done we should be ready to try this out.

Restart the server, then go ahead and make the following request with a `page_size=2` parameter:

```
$ curl "localhost:4000/v1/movies?page_size=2"
{
  "movies": [
    {
      "id": 1,
      "title": "Moana",
      "year": 2016,
      "runtime": "107 mins",
      "genres": [
        "animation",
        "adventure"
      ],
      "version": 1
    },
    {
      "id": 2,
      "title": "Black Panther",
      "year": 2018,
      "runtime": "134 mins",
      "genres": [
        "sci-fi",
        "action",
        "adventure"
      ],
      "version": 2
    }
  ]
}
```

That's looking good. Our endpoint is now returning the first two movie records only from our database (using the default sort order of ascending movie ID).

Then let's try requesting the second page of results. If you've been following along, this page should include the one remaining record in our system, like so:

```
# IMPORTANT: This URL must be surrounded with double-quotes to work correctly.
$ curl "localhost:4000/v1/movies?page_size=2&page=2"
{
  "movies": [
    {
      "id": 4,
      "title": "The Breakfast Club",
      "year": 1985,
      "runtime": "97 mins",
      "genres": [
        "comedy"
      ],
      "version": 5
    }
  ]
}
```

If you try to request the third page, you should get an empty JSON array in the response like so:


```
$ curl "localhost:4000/v1/movies?page_size=2&page=3"
{
  "movies": []
}
```

Returning Pagination Metadata

At this point the pagination on our `GET /v1/movies` endpoint is working nicely, but it would be even better if we could include some additional metadata along with the response. Information like the *current* and *last* page numbers, and the *total number of available records* would help to give the client context about the response and make navigating through the pages easier.

In this chapter we'll improve the response so that it includes additional pagination metadata, similar to this:

```
{
  "metadata": {
    "current_page": 1,
    "page_size": 20,
    "first_page": 1,
    "last_page": 42,
    "total_records": 832
  },
  "movies": [
    {
      "id": 1,
      "title": "Moana",
      "year": 2015,
      "runtime": "107 mins",
      "genres": [
        "animation",
        "adventure"
      ],
      "version": 1
    },
    ...
  ]
}
```

Calculating the total records

The challenging part of doing this is generating the `total_records` figure. We want this to reflect the total number of available records *given the title and genres filters that are applied* — not the absolute total of records in the `movies` table.

A neat way to do this is to adapt our existing SQL query to include a [window function](#) which counts the total number of filtered rows, like so:

```
SELECT count(*) OVER(), id, created_at, title, year, runtime, genres, version
FROM movies
WHERE (to_tsvector('simple', title) @@ plainto_tsquery('simple', $1) OR $1 = '')
AND (genres @> $2 OR $2 = '{}')
ORDER BY %s %s, id ASC
LIMIT $3 OFFSET $4
```

The inclusion of the `count(*) OVER()` expression at the start of the query will result in the filtered record count being included as the first value in each row. A bit like this:

count	id	created_at	title	year	runtime	genres	version
3	1	2020-11-27 17:17:25+01	Moana	2015	107	{animation,adventure}	1
3	2	2020-11-27 18:01:45+01	Black Panther	2018	134	{sci-fi,action,adventure}	2
3	4	2020-11-27 18:02:20+01	The Breakfast Club	1985	97	{comedy,drama}	6

When PostgreSQL executes this SQL query, the (very simplified) sequence of events runs broadly like this:

1. The **WHERE** clause is used to filter the data in the `movies` table and get the *qualifying rows*.
2. The window function `count(*) OVER()` is applied, which counts all the qualifying rows.
3. The **ORDER BY** rules are applied and the qualifying rows are sorted.
4. The **LIMIT** and **OFFSET** rules are applied and the appropriate sub-set of sorted qualifying rows is returned.

Updating the code

With that brief explanation out of the way, let's get this up and running. We'll begin by updating the `internal/data/filters.go` file to define a new `Metadata` struct to hold the pagination metadata, along with a helper to calculate the values. Like so:

File: internal/data/filters.go

```
package data

import (
    "math" // New import
    "strings"

    "greenlight.alexedwards.net/internal/validator"
)

...

// Define a new Metadata struct for holding the pagination metadata.
type Metadata struct {
    CurrentPage int `json:"current_page,omitempty"`
    PageSize    int `json:"page_size,omitempty"`
    FirstPage   int `json:"first_page,omitempty"`
    LastPage    int `json:"last_page,omitempty"`
    TotalRecords int `json:"total_records,omitempty"`
}

// The calculateMetadata() function calculates the appropriate pagination metadata
// values given the total number of records, current page, and page size values. Note
// that the last page value is calculated using the math.Ceil() function, which rounds
// up a float to the nearest integer. So, for example, if there were 12 records in total
// and a page size of 5, the last page value would be math.Ceil(12/5) = 3.
func calculateMetadata(totalRecords, page, pageSize int) Metadata {
    if totalRecords == 0 {
        // Note that we return an empty Metadata struct if there are no records.
        return Metadata{}
    }

    return Metadata{
        CurrentPage: page,
        PageSize:    pageSize,
        FirstPage:   1,
        LastPage:    int(math.Ceil(float64(totalRecords) / float64(pageSize))),
        TotalRecords: totalRecords,
    }
}
```

Then we need to head back to our `GetAll()` method and update it to use our new SQL query (with the window function) to get the total records count. Then, if everything works successfully, we'll use the `calculateMetadata()` function to generate the pagination metadata and return it alongside the movie data.

Go ahead and update the `GetAll()` function like so:

File: internal/data/movies.go

```
package data

...

// Update the function signature to return a Metadata struct.
func (m MovieModel) GetAll(title string, genres []string, filters Filters) ([]*Movie, Metadata, error) {
    // Update the SQL query to include the window function which counts the total
    // (filtered) records.
```

```

// ...
query := fmt.Sprintf(`
    SELECT count(*) OVER(), id, created_at, title, year, runtime, genres, version
    FROM movies
    WHERE (to_tsvector('simple', title) @@ plainto_tsquery('simple', $1) OR $1 = '')
    AND (genres @> $2 OR $2 = '{}')
    ORDER BY %s %s, id ASC
    LIMIT $3 OFFSET $4`, filters.sortColumn(), filters.sortDirection())

ctx, cancel := context.WithTimeout(context.Background(), 3*time.Second)
defer cancel()

args := []interface{}{title, pq.Array(genres), filters.limit(), filters.offset()}

rows, err := m.DB.QueryContext(ctx, query, args...)
if err != nil {
    return nil, Metadata{}, err // Update this to return an empty Metadata struct.
}

defer rows.Close()

// Declare a totalRecords variable.
totalRecords := 0
movies := []*Movie{}

for rows.Next() {
    var movie Movie

    err := rows.Scan(
        &totalRecords, // Scan the count from the window function into totalRecords.
        &movie.ID,
        &movie.CreatedAt,
        &movie.Title,
        &movie.Year,
        &movie.Runtime,
        pq.Array(&movie.Genres),
        &movie.Version,
    )
    if err != nil {
        return nil, Metadata{}, err // Update this to return an empty Metadata struct.
    }

    movies = append(movies, &movie)
}

if err = rows.Err(); err != nil {
    return nil, Metadata{}, err // Update this to return an empty Metadata struct.
}

// Generate a Metadata struct, passing in the total record count and pagination
// parameters from the client.
metadata := calculateMetadata(totalRecords, filters.Page, filters.PageSize)

// Include the metadata struct when returning.
return movies, metadata, nil
}

```

Finally, we need to update our `listMoviesHandler` handler to receive the `Metadata` struct returned by `GetAll()` and include the information in the JSON response for the client. Like so:

File: cmd/api/movies.go

```
package main

...

func (app *application) listMoviesHandler(w http.ResponseWriter, r *http.Request) {
    var input struct {
        Title string
        Genres []string
        data.Filters
    }

    v := validator.New()

    qs := r.URL.Query()

    input.Title = app.readString(qs, "title", "")
    input.Genres = app.readCSV(qs, "genres", []string{})

    input.Filters.Page = app.readInt(qs, "page", 1, v)
    input.Filters.PageSize = app.readInt(qs, "page_size", 20, v)

    input.Filters.Sort = app.readString(qs, "sort", "id")
    input.Filters.SortSafelist = []string{"id", "title", "year", "runtime", "-id", "-title", "-year", "-runtime"}

    if data.ValidateFilters(v, input.Filters); !v.Valid() {
        app.failedValidationResponse(w, r, v.Errors)
        return
    }

    // Accept the metadata struct as a return value.
    movies, metadata, err := app.models.Movies.GetAll(input.Title, input.Genres, input.Filters)
    if err != nil {
        app.serverErrorResponse(w, r, err)
        return
    }

    // Include the metadata in the response envelope.
    err = app.writeJSON(w, http.StatusOK, envelope{"movies": movies, "metadata": metadata}, nil)
    if err != nil {
        app.serverErrorResponse(w, r, err)
    }
}
```

Feel free to restart the API and try out this new functionality by making some different requests to the `GET /v1/movies` endpoint. You should find that the correct pagination metadata is now included in the response. For example:

```
$ curl "localhost:4000/v1/movies?page=1&page_size=2"
{
  "metadata": {
    "current_page": 1,
    "page_size": 2,
    "first_page": 1,
    "last_page": 2,
    "total_records": 3
  },
  "movies": [
    {
      "id": 1,
      "title": "Moana",
      "year": 2015,
      "runtime": "107 mins",
      "genres": [
        "animation",
        "adventure"
      ],
      "version": 1
    },
    {
      "id": 2,
      "title": "Black Panther",
      "year": 2018,
      "runtime": "134 mins",
      "genres": [
        "sci-fi",
        "action",
        "adventure"
      ],
      "version": 2
    }
  ]
}
```

And if you try making a request with a filter applied, you should see that the `last_page` value and `total_records` count changes to reflect the applied filters. For example, by requesting only movies with the genre “adventure” we can see that the `total_records` count drops down to `2`:

```
$ curl localhost:4000/v1/movies?genres=adventure
{
  "metadata": {
    "current_page": 1,
    "page_size": 20,
    "first_page": 1,
    "last_page": 1,
    "total_records": 2
  },
  "movies": [
    {
      "id": 1,
      "title": "Moana",
      "year": 2015,
      "runtime": "107 mins",
      "genres": [
        "animation",
        "adventure"
      ],
      "version": 1
    },
    {
      "id": 2,
      "title": "Black Panther",
      "year": 2018,
      "runtime": "134 mins",
      "genres": [
        "sci-fi",
        "action",
        "adventure"
      ],
      "version": 2
    }
  ]
}
```

Lastly, if you make a request with a too-high page value, you should get a response with an empty metadata object and movies array, like this:

```
$ curl localhost:4000/v1/movies?page=100
{
  "metadata": {},
  "movies": []
}
```

Over the last few chapters, we've had to put in a lot of work on the `GET /v1/movies` endpoint. But the end result is really powerful. The client now has a lot of control over what their response contains, with filtering, pagination and sorting all supported.

With the `Filters` struct that we've created, we've also got something that we can easily drop into any other endpoints that need pagination and sorting functionality in the future. And if you take a step back and look at the final code we've written in the `listMovieHandler` and our database model's `GetAll()` method, there isn't *so much* more code than there was in the earlier iterations of the endpoint.

Structured Logging and Error Handling

In this section of the book, we're going to take a short break from working on our API endpoints and focus on making some improvements to the way that we handle exceptions and errors.

You'll learn how to:

- Create a custom logger for writing *structured, levelled, log messages* in JSON format.
- Integrate the custom logger with your handlers and Go's `http.Server`, so that all log messages are written to one location in a consistent format.
- How to recover and gracefully handle panics in your application middleware and handlers.

Structured JSON Log Entries

At the moment, all of our log entries are simple free-form messages prefixed by the current date and time, written to the *standard out* stream. A good example of these are the log entries that we see when we start the API:

```
$ go run ./cmd/api
2021/04/09 19:43:31 database connection pool established
2021/04/09 19:43:31 starting development server on :4000
```

For many applications this simple style of logging will be *good enough*, and there's no need to introduce additional complexity to your project.

But for applications which do a lot of logging, it can often be useful to enforce a consistent structure and format for your log entries. This can help make it easier to distinguish between different types of log entry (like *informational* and *error* entries), easier to search and filter log entries, and simpler to integrate your logs with third-party analysis and monitoring systems.

So in this chapter we're going demonstrate how to adapt our application so that log entries are written in a structured JSON format, similar to this:

```
{"level":"INFO","time":"2020-12-16T10:53:35Z","message":"starting server","properties":{"addr":":4000","env":"development"}}
```

Essentially, each log entry will be a single JSON object containing the following key/value pairs:

Key	Description
level	A code indicating the severity of the log entry. In this project we will use the following three severity levels, ordered from least to most severe: - INFO (least severe) - ERROR - FATAL (most severe)
time	The UTC time that the log entry was made with second precision.
message	A string containing the free-text information or error message.
properties	Any additional information relevant to the log entry in string key/value pairs (optional).
trace	A stack trace for debugging purposes (optional).

Creating a custom logger

To implement this in our API, we're going to create our own custom **Logger** type which writes structured log entries in JSON format.

We'll implement some convenience methods on **Logger**, such as **PrintInfo()** and **PrintError()**, in order to make it easy to write log entries with a specific level from our application in a single command. And we'll also design the logger so that only entries at or above a certain *minimum severity level* are actually logged — so if you only want to record **ERROR** and **FATAL** messages (and not **INFO** messages), then that is easy to do.

If you're following along, go ahead and create a new **internal/jsonlog** package like so:

```
$ mkdir internal/jsonlog
$ touch internal/jsonlog/jsonlog.go
```

And then add the following code:

File: internal/jsonlog/jsonlog.go

```
package jsonlog

import (
    "encoding/json"
    "io"
    "os"
    "runtime/debug"
    "sync"
    "time"
)
```

```

// Define a Level type to represent the severity level for a log entry.
type Level int8

// Initialize constants which represent a specific severity level. We use the iota
// keyword as a shortcut to assign successive integer values to the constants.
const (
    LevelInfo  Level = iota // Has the value 0.
    LevelError           // Has the value 1.
    LevelFatal           // Has the value 2.
    LevelOff             // Has the value 3.
)

// Return a human-friendly string for the severity level.
func (l Level) String() string {
    switch l {
    case LevelInfo:
        return "INFO"
    case LevelError:
        return "ERROR"
    case LevelFatal:
        return "FATAL"
    default:
        return ""
    }
}

// Define a custom Logger type. This holds the output destination that the log entries
// will be written to, the minimum severity level that log entries will be written for,
// plus a mutex for coordinating the writes.
type Logger struct {
    out      io.Writer
    minLevel Level
    mu       sync.Mutex
}

// Return a new Logger instance which writes log entries at or above a minimum severity
// level to a specific output destination.
func New(out io.Writer, minLevel Level) *Logger {
    return &Logger{
        out:      out,
        minLevel: minLevel,
    }
}

// Declare some helper methods for writing log entries at the different levels. Notice
// that these all accept a map as the second parameter which can contain any arbitrary
// 'properties' that you want to appear in the log entry.
func (l *Logger) PrintInfo(message string, properties map[string]string) {
    l.print(LevelInfo, message, properties)
}

func (l *Logger) PrintError(err error, properties map[string]string) {
    l.print(LevelError, err.Error(), properties)
}

func (l *Logger) PrintFatal(err error, properties map[string]string) {
    l.print(LevelFatal, err.Error(), properties)
    os.Exit(1) // For entries at the FATAL level, we also terminate the application.
}

// Print is an internal method for writing the log entry.
func (l *Logger) print(level Level, message string, properties map[string]string) (int, error) {
    // If the severity level of the log entry is below the minimum severity for the
    // logger, then return with no further action.
    if level < l.minLevel {
        return 0, nil
    }

```

```

    return v, nil
}

// Declare an anonymous struct holding the data for the log entry.
aux := struct {
    Level      string      `json:"level"`
    Time       string      `json:"time"`
    Message    string      `json:"message"`
    Properties map[string]string `json:"properties,omitempty"`
    Trace      string      `json:"trace,omitempty"`
}{
    Level:      level.String(),
    Time:       time.Now().UTC().Format(time.RFC3339),
    Message:    message,
    Properties: properties,
}

// Include a stack trace for entries at the ERROR and FATAL levels.
if level >= LevelError {
    aux.Trace = string(debug.Stack())
}

// Declare a line variable for holding the actual log entry text.
var line []byte

// Marshal the anonymous struct to JSON and store it in the line variable. If there
// was a problem creating the JSON, set the contents of the log entry to be that
// plain-text error message instead.
line, err := json.Marshal(aux)
if err != nil {
    line = []byte(LevelError.String() + ": unable to marshal log message: " + err.Error())
}

// Lock the mutex so that no two writes to the output destination can happen
// concurrently. If we don't do this, it's possible that the text for two or more
// log entries will be intermingled in the output.
l.mu.Lock()
defer l.mu.Unlock()

// Write the log entry followed by a newline.
return l.out.Write(append(line, '\n'))
}

// We also implement a Write() method on our Logger type so that it satisfies the
// io.Writer interface. This writes a log entry at the ERROR level with no additional
// properties.
func (l *Logger) Write(message []byte) (n int, err error) {
    return l.print(LevelError, string(message), nil)
}

```

Note: If you want, you could extend this code to support additional [severity levels](#) such as `DEBUG` and `WARNING`.

Essentially, our `Logger` type is a fairly thin wrapper around an `io.Writer`. We have some helper methods like `PrintInfo()` and `PrintError()` which accept some data for the log entry, encode this data to JSON, and then write it to the `io.Writer`.

The two most interesting things to point out in this code are our use of the `iota`

enumerator to create the log levels and a `sync.Mutex` to coordinate the writes.

You can use `iota` to easily assign successive integer values to a set of integer *constants*. It starts at zero, and increments by 1 for every constant declaration, resetting to 0 when the word `const` appears in the code again. The way that we've used this means that each of our severity levels has an integer value, with the *more severe levels having a higher value*.

We're also using a `sync.Mutex` (a *mutual exclusion lock*) to prevent our `Logger` instance making multiple writes concurrently. Without this mutex lock, it's possible that the content of multiple log entries would be written at exactly the same time and be mixed up in the output, rather than each entry being written in full on its own line.

Important: How mutexes work, and how to use them, can be quite confusing if you haven't encountered them before and it's impossible to fully explain in a few short sentences. I've written a much more detailed article — [Understanding Mutexes](#) — which provides a proper explanation, and if you're not already confident with mutexes I highly recommend reading this before you continue.

Now that's in place, let's update our `cmd/api/main.go` file to create a new `Logger` instance and then use it in our code. We'll also add it to our `application` struct, so that it's available as a dependency to all our handlers and helpers.

Like so:

```
File: cmd/api/main.go

package main

import (
    "context"
    "database/sql"
    "flag"
    "fmt"
    "net/http"
    "os"
    "time"

    "greenlight.alexedwards.net/internal/data"
    "greenlight.alexedwards.net/internal/jsonlog" // New import

    _ "github.com/lib/pq"
)

...

// Change the logger field to have the type *jsonlog.Logger, instead of
// *log.Logger.
type application struct {
    config config
    logger *jsonlog.Logger
}
```

```

models data.Models
}

func main() {
    var cfg config

    flag.IntVar(&cfg.port, "port", 4000, "API server port")
    flag.StringVar(&cfg.env, "env", "development", "Environment (development|staging|production)")

    flag.StringVar(&cfg.db.dsn, "db-dsn", os.Getenv("GREENLIGHT_DB_DSN"), "PostgreSQL DSN")

    flag.IntVar(&cfg.db.maxOpenConns, "db-max-open-conns", 25, "PostgreSQL max open connections")
    flag.IntVar(&cfg.db.maxIdleConns, "db-max-idle-conns", 25, "PostgreSQL max idle connections")
    flag.StringVar(&cfg.db.maxIdleTime, "db-max-idle-time", "15m", "PostgreSQL max connection idle time")

    flag.Parse()

    // Initialize a new jsonlog.Logger which writes any messages *at or above* the INFO
    // severity level to the standard out stream.
    logger := jsonlog.New(os.Stdout, jsonlog.LevelInfo)

    db, err := openDB(cfg)
    if err != nil {
        // Use the PrintFatal() method to write a log entry containing the error at the
        // FATAL level and exit. We have no additional properties to include in the log
        // entry, so we pass nil as the second parameter.
        logger.PrintFatal(err, nil)
    }

    defer db.Close()
    // Likewise use the PrintInfo() method to write a message at the INFO level.
    logger.PrintInfo("database connection pool established", nil)

    app := &application{
        config: cfg,
        logger: logger,
        models: data.NewModels(db),
    }

    srv := &http.Server{
        Addr:         fmt.Sprintf(":%d", cfg.port),
        Handler:      app.routes(),
        IdleTimeout:  time.Minute,
        ReadTimeout:  10 * time.Second,
        WriteTimeout: 30 * time.Second,
    }

    // Again, we use the PrintInfo() method to write a "starting server" message at the
    // INFO level. But this time we pass a map containing additional properties (the
    // operating environment and server address) as the final parameter.
    logger.PrintInfo("starting server", map[string]string{
        "addr": srv.Addr,
        "env":  cfg.env,
    })

    err = srv.ListenAndServe()
    // Use the PrintFatal() method to log the error and exit.
    logger.PrintFatal(err, nil)
}

...

```

We'll also need to update the `logError()` helper in our `cmd/api/errors.go` file to use our

new **Logger** instance. One of the nice things here is that it's simple to include additional information as properties when logging the error.

If you're following along, go ahead and update the **logError()** method so that it records the request URL and method alongside the error message. Like so:

```
File: cmd/api/errors.go

package main

...

func (app *application) logError(r *http.Request, err error) {
    // Use the PrintError() method to log the error message, and include the current
    // request method and URL as properties in the log entry.
    app.logger.PrintError(err, map[string]string{
        "request_method": r.Method,
        "request_url":    r.URL.String(),
    })
}

...
```

At this point, if you restart the application you should now see two **INFO** level log entries in JSON format like so:

```
$ go run ./cmd/api
{"level":"INFO","time":"2021-04-10T08:08:36Z","message":"database connection pool established"}
{"level":"INFO","time":"2021-04-10T08:08:36Z","message":"starting server","properties":{"addr":":4000","env":"development"}}
```

Notice how in the second log entry the additional properties have been included in a nested JSON object?

If you like, you can also try opening a second terminal window and starting another instance of the API listening on the same port. This should result in a **FATAL** log entry and stack trace being printed, followed by the application immediately exiting. Similar to this:

```
$ go run ./cmd/api
{"level":"INFO","time":"2021-04-10T08:09:53Z","message":"database connection pool established"}
{"level":"INFO","time":"2021-04-10T08:09:53Z","message":"starting server","properties":{"addr":":4000","env":"development"}}
{"level":"FATAL","time":"2021-04-10T08:09:53Z","message":"listen tcp :4000: bind: address already in use","trace":"..."}
exit status 1
```

Additional Information

Integration with the `http.Server` error log

It's important to be aware that Go's `http.Server` may also write its own log messages relating to things like unrecovered panics, or problems accepting and writing to HTTP connections.

By default, it writes these messages to the [standard logger](#) — which means they will be written to the standard error stream (instead of standard out like our other log messages), and they won't be in our nice new JSON format.

If you want everything logged in one place, in the same format, then that's obviously a problem.

Unfortunately, you can't set `http.Server` to use our new `Logger` type directly. Instead, you will need to leverage the fact that our `Logger` satisfies the `io.Writer` interface (thanks to the `Write()` method that we added to it), and set `http.Server` to use a regular `log.Logger` instance from the standard library which *writes to* our own `Logger` as the target destination.

It's simpler than it sounds. All you have to do is something like this:

```
func main() {  
  
    ...  
  
    // Initialize the custom logger.  
    logger := jsonlog.New(os.Stdout, jsonlog.LevelInfo)  
  
    ...  
  
    srv := &http.Server{  
        Addr:      fmt.Sprintf(":%d", cfg.port),  
        Handler:    app.routes(),  
        // Create a new Go log.Logger instance with the log.New() function, passing in  
        // our custom Logger as the first parameter. The "" and 0 indicate that the  
        // log.Logger instance should not use a prefix or any flags.  
        ErrorLog:    log.New(logger, "", 0),  
        IdleTimeout: time.Minute,  
        ReadTimeout: 10 * time.Second,  
        WriteTimeout: 30 * time.Second,  
    }  
  
    ...  
}
```

With that set up, any log messages that `http.Server` writes will be passed to our `Logger.Write()` method, which in turn will output a log entry in JSON format at the `ERROR` level.

Third-party logging packages

If you don't want to implement your own custom logger, like we have in this chapter, then there is a plethora of [third-party packages](#) available which implement structured logging in JSON (and other) formats.

If I had to recommend one, it would be [zerolog](#). It has a nice interface, and a good range of customization options. It's also designed to be very fast and cheap in terms of memory allocations when writing log entries, so if your application is doing a lot of logging (i.e. more than just occasional information and error messages) then it may be a sensible choice.

Panic Recovery

At the moment any panics in our API handlers will be recovered automatically by Go's `http.Server`. This will unwind the stack for the affected goroutine (calling any deferred functions along the way), close the underlying HTTP connection, and log an error message and stack trace.

This behavior is *OK*, but it would be better for the client if we could also send a `500 Internal Server Error` response to explain that something has gone wrong — rather than just closing the HTTP connection with no context.

In *Let's Go* we talked through the details of how to do this by creating some middleware to *recover* the panic, and it makes sense to do the same thing again here.

If you're following along, go ahead and create a `cmd/api/middleware.go` file:

```
$ touch cmd/api/middleware.go
```

And inside that file add a new `recoverPanic()` middleware:

File: cmd/api/middleware.go

```
package main

import (
    "fmt"
    "net/http"
)

func (app *application) recoverPanic(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        // Create a deferred function (which will always be run in the event of a panic
        // as Go unwinds the stack).
        defer func() {
            // Use the builtin recover function to check if there has been a panic or
            // not.
            if err := recover(); err != nil {
                // If there was a panic, set a "Connection: close" header on the
                // response. This acts as a trigger to make Go's HTTP server
                // automatically close the current connection after a response has been
                // sent.
                w.Header().Set("Connection", "close")
                // The value returned by recover() has the type interface{}, so we use
                // fmt.Errorf() to normalize it into an error and call our
                // serverErrorResponse() helper. In turn, this will log the error using
                // our custom Logger type at the ERROR level and send the client a 500
                // Internal Server Error response.
                app.serverErrorResponse(w, r, fmt.Errorf("%s", err))
            }
        }()

        next.ServeHTTP(w, r)
    })
}
```

Once that's done, we need to update our `cmd/api/routes.go` file so that the `recoverPanic()` middleware wraps our router. This will ensure that the middleware runs for every one of our API endpoints.

File: cmd/api/routes.go

```
package main

...

// Update the routes() method to return a http.Handler instead of a *httprouter.Router.
func (app *application) routes() http.Handler {
    router := httprouter.New()

    router.NotFound = http.HandlerFunc(app.notFoundResponse)
    router.MethodNotAllowed = http.HandlerFunc(app.methodNotAllowedResponse)

    router.HandleFunc(http.MethodGet, "/v1/healthcheck", app.healthcheckHandler)

    router.HandleFunc(http.MethodGet, "/v1/movies", app.listMoviesHandler)
    router.HandleFunc(http.MethodPost, "/v1/movies", app.createMovieHandler)
    router.HandleFunc(http.MethodGet, "/v1/movies/:id", app.showMovieHandler)
    router.HandleFunc(http.MethodPatch, "/v1/movies/:id", app.updateMovieHandler)
    router.HandleFunc(http.MethodDelete, "/v1/movies/:id", app.deleteMovieHandler)

    // Wrap the router with the panic recovery middleware.
    return app.recoverPanic(router)
}
```

Now that's in place, if there is a panic in one of our API handlers the `recoverPanic()` middleware will recover it and call our regular `app.serverErrorResponse()` helper. In turn, that will log the error using our custom `Logger` and send the client a nice `500 Internal Server Error` response with a JSON body.

Additional Information

Panic recovery in other goroutines

It's really important to realize that our middleware will only recover panics that happen in the *same goroutine that executed the `recoverPanic()` middleware*.

If, for example, you have a handler which spins up another goroutine (e.g. to do some background processing), then any panics that happen in the background goroutine will not be recovered — not by the `recoverPanic()` middleware... and not by the panic recovery built into `http.Server`. These panics will cause your application to exit and bring down the server.

So, if you are spinning up additional goroutines from within your handlers and there is any chance of a panic, you *must* make sure that you recover any panics from within those goroutines too.

We'll look at this topic in more detail later in the book, and demonstrate how to deal with it when we use a background goroutine to send welcome emails to our API users.

Rate Limiting

If you're building an API for public use, then it's quite likely that you'll want to implement some form of *rate limiting* to prevent clients from making *too many requests too quickly*, and putting excessive strain on your server.

In this section of the book we're going to create some middleware to help with that.

Essentially, we want this middleware to check how many requests have been received in the last 'N' seconds and — if there have been too many — then it should send the client a **429 Too Many Requests** response. We'll position this middleware before our main application handlers, so that it carries out this check *before* we do any expensive processing like decoding a JSON request body or querying our database.

You'll learn:

- About the principles behind *token-bucket* rate-limiter algorithms and how we can apply them in the context of an API or web application.
- How to create middleware to rate-limit requests to your API endpoints, first by making a single rate global limiter, then extending it to support per-client limiting based on IP address.
- How to make rate limiter behavior configurable at runtime, including disabling the rate limiter altogether for testing purposes.

Global Rate Limiting

Let's build things up slowly and start by creating a single *global rate limiter* for our application. This will consider *all the requests* that our API receives (rather than having separate rate limiters for every individual client).

Instead of writing our own rate-limiting logic from scratch, which would be quite complex and time-consuming, we can leverage the [x/time/rate](#) package to help us here. This provides a tried-and-tested implementation of a *token bucket* rate limiter.

If you're following along, please go ahead and download the latest version of this package like so:

```
$ go get golang.org/x/time/rate@latest
go: downloading golang.org/x/time v0.0.0-20210723032227-1f47c861a9ac
go: added golang.org/x/time v0.0.0-20210723032227-1f47c861a9ac
```

Before we start writing any code, let's take a moment to explain how token-bucket rate limiters work. The description from the official [x/time/rate](#) documentation says:

*A Limiter controls how frequently events are allowed to happen. It implements a “token bucket” of size **b**, initially full and refilled at rate **r** tokens per second.*

Putting that into the context of our API application...

- We will have a bucket that starts with **b** tokens in it.
- Each time we receive a HTTP request, we will remove one token from the bucket.
- Every $1/r$ seconds, a token is added back to the bucket — up to a maximum of **b** total tokens.
- If we receive a HTTP request and the bucket is empty, then we should return a **429 Too Many Requests** response.

In practice this means that our application would allow a maximum ‘burst’ of **b** HTTP requests in quick succession, but over time it would allow an average of **r** requests per second.

In order to create a token bucket rate limiter from [x/time/rate](#), we will need to use the `NewLimiter()` function. This has a signature which looks like this:


```
// Note that the Limit type is an 'alias' for float64.  
func NewLimiter(r Limit, b int) *Limiter
```

So if we want to create a rate limiter which allows an average of 2 requests per second, with a maximum of 4 requests in a single ‘burst’, we could do so with the following code:

```
// Allow 2 requests per second, with a maximum of 4 requests in a burst.  
limiter := rate.NewLimiter(2, 4)
```

Enforcing a global rate limit

OK, with that high-level explanation out of the way, let’s jump into some code and see how this works in practice.

One of the nice things about the middleware pattern that we are using is that it is straightforward to include ‘initialization’ code which only runs once when we *wrap* something with the middleware, rather than running on every request that the middleware handles.

```
func (app *application) exampleMiddleware(next http.Handler) http.Handler {  
  
    // Any code here will run only once, when we wrap something with the middleware.  
  
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {  
  
        // Any code here will run for every request that the middleware handles.  
  
        next.ServeHTTP(w, r)  
    })  
}
```

In our case, we’ll make a new `rateLimit()` middleware method which creates a new rate limiter as part of the ‘initialization’ code, and then uses this rate limiter for every request that it subsequently handles.

If you’re following along, open up the `cmd/api/middleware.go` file and create the middleware like so:

File: cmd/api/middleware.go

```
package main

import (
    "fmt"
    "net/http"

    "golang.org/x/time/rate" // New import
)

...

func (app *application) rateLimit(next http.Handler) http.Handler {
    // Initialize a new rate limiter which allows an average of 2 requests per second,
    // with a maximum of 4 requests in a single 'burst'.
    limiter := rate.NewLimiter(2, 4)

    // The function we are returning is a closure, which 'closes over' the limiter
    // variable.
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        // Call limiter.Allow() to see if the request is permitted, and if it's not,
        // then we call the rateLimitExceededResponse() helper to return a 429 Too Many
        // Requests response (we will create this helper in a minute).
        if !limiter.Allow() {
            app.rateLimitExceededResponse(w, r)
            return
        }

        next.ServeHTTP(w, r)
    })
}
```

In this code, whenever we call the `Allow()` method on the rate limiter exactly one token will be consumed from the bucket. If there are no tokens left in the bucket, then `Allow()` will return `false` and that acts as the trigger for us send the client a `429 Too Many Requests` response.

It's also important to note that the code behind the `Allow()` method is protected by a mutex and is safe for concurrent use.

Let's now go to our `cmd/api/errors.go` file and create the `rateLimitExceededResponse()` helper. Like so:

File: cmd/api/errors.go

```
package main

...

func (app *application) rateLimitExceededResponse(w http.ResponseWriter, r *http.Request) {
    message := "rate limit exceeded"
    app.errorResponse(w, r, http.StatusTooManyRequests, message)
}
```

Then, lastly, in the `cmd/api/routes.go` file we want to add the `rateLimit()` middleware to our middleware chain. This should come after our panic recovery middleware (so that any panics in `rateLimit()` are recovered), but otherwise we want it to be used as early as possible to prevent unnecessary work for our server.

Go ahead and update the file accordingly:

File: `cmd/api/routes.go`

```
package main

...

func (app *application) routes() http.Handler {
    router := httprouter.New()

    router.NotFound = http.HandlerFunc(app.notFoundResponse)
    router.MethodNotAllowed = http.HandlerFunc(app.methodNotAllowedResponse)

    router.HandleFunc(http.MethodGet, "/v1/healthcheck", app.healthcheckHandler)

    router.HandleFunc(http.MethodGet, "/v1/movies", app.listMoviesHandler)
    router.HandleFunc(http.MethodPost, "/v1/movies", app.createMovieHandler)
    router.HandleFunc(http.MethodGet, "/v1/movies/:id", app.showMovieHandler)
    router.HandleFunc(http.MethodPatch, "/v1/movies/:id", app.updateMovieHandler)
    router.HandleFunc(http.MethodDelete, "/v1/movies/:id", app.deleteMovieHandler)

    // Wrap the router with the rateLimit() middleware.
    return app.recoverPanic(app.rateLimit(router))
}
```

Now we should be ready to try this out!

Restart the API, then in another terminal window execute the following command to issue a batch of 6 requests to our `GET /v1/healthcheck` endpoint in quick succession. You should get responses which look like this:

```
$ for i in {1..6}; do curl http://localhost:4000/v1/healthcheck; done
{
  "status": "available",
  "system_info": {
    "environment": "development",
    "version": "1.0.0"
  }
}
{
  "status": "available",
  "system_info": {
    "environment": "development",
    "version": "1.0.0"
  }
}
{
  "status": "available",
  "system_info": {
    "environment": "development",
    "version": "1.0.0"
  }
}
{
  "status": "available",
  "system_info": {
    "environment": "development",
    "version": "1.0.0"
  }
}
{
  "error": "rate limit exceeded"
}
{
  "error": "rate limit exceeded"
}
```

We can see from this that the first 4 requests succeed, due to our limiter being set up to permit a 'burst' of 4 requests in quick succession. But once those 4 requests were used up, the tokens in the bucket ran out and our API began to return the **"rate limit exceeded"** error response instead.

If you wait a second and re-run this command, you should find that some requests in the second batch succeed again, due to the token bucket being refilled at the rate of two tokens every second.

IP-based Rate Limiting

Using a global rate limiter can be useful when you want to enforce a strict limit on the total rate of requests to your API, and you don't care where the requests are coming from. But it's generally more common to want a separate rate limiter for each client, so that one bad client making too many requests doesn't affect all the others.

A conceptually straightforward way to implement this is to create an in-memory *map of rate limiters*, using the IP address for each client as the map key.

Each time a new client makes a request to our API, we will initialize a new rate limiter and add it to the map. For any subsequent requests, we will retrieve the client's rate limiter from the map and check whether the request is permitted by calling its `Allow()` method, just like we did before.

Because we'll potentially have multiple goroutines accessing the map concurrently, we'll need to protect access to the map by using a mutex to prevent race conditions.

If you're following along, let's jump into the code and update our `rateLimit()` middleware to implement this.

File: cmd/api/middleware.go

```
package main

import (
    "fmt"
    "net" // New import
    "net/http"
    "sync" // New import

    "golang.org/x/time/rate"
)

...

func (app *application) ratelimit(next http.Handler) http.Handler {
    // Declare a mutex and a map to hold the clients' IP addresses and rate limiters.
    var (
        mu      sync.Mutex
        clients = make(map[string]*rate.Limiter)
    )

    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        // Extract the client's IP address from the request.
        ip, _, err := net.SplitHostPort(r.RemoteAddr)
        if err != nil {
            app.serverErrorResponse(w, r, err)
            return
        }

        // Lock the mutex to prevent this code from being executed concurrently.
        mu.Lock()

        // Check to see if the IP address already exists in the map. If it doesn't, then
        // initialize a new rate limiter and add the IP address and limiter to the map.
        if _, found := clients[ip]; !found {
            clients[ip] = rate.NewLimiter(2, 4)
        }

        // Call the Allow() method on the rate limiter for the current IP address. If
        // the request isn't allowed, unlock the mutex and send a 429 Too Many Requests
        // response, just like before.
        if !clients[ip].Allow() {
            mu.Unlock()
            app.rateLimitExceededResponse(w, r)
            return
        }

        // Very importantly, unlock the mutex before calling the next handler in the
        // chain. Notice that we DON'T use defer to unlock the mutex, as that would mean
        // that the mutex isn't unlocked until all the handlers downstream of this
        // middleware have also returned.
        mu.Unlock()

        next.ServeHTTP(w, r)
    })
}
```

Deleting old limiters

The code above will work, but there's a slight problem — the `clients` map will grow indefinitely, taking up more and more resources with every new IP address and rate limiter that we add.

To prevent this, let's update our code so that we also record the *last seen* time for each client. We can then run a background goroutine in which we periodically delete any clients that we haven't been seen recently from the `clients` map.

To make this work, we'll need to create a custom `client` struct which holds both the rate limiter and last seen time for each client, and launch the background cleanup goroutine when initializing the middleware.

Like so:

File: cmd/api/middleware.go

```
package main

import (
    "fmt"
    "net"
    "net/http"
    "sync"
    "time" // New import

    "golang.org/x/time/rate"
)

...

func (app *application) ratelimit(next http.Handler) http.Handler {
    // Define a client struct to hold the rate limiter and last seen time for each
    // client.
    type client struct {
        limiter *rate.Limiter
        lastSeen time.Time
    }

    var (
        mu sync.Mutex
        // Update the map so the values are pointers to a client struct.
        clients = make(map[string]*client)
    )

    // Launch a background goroutine which removes old entries from the clients map once
    // every minute.
    go func() {
        for {
            time.Sleep(time.Minute)

            // Lock the mutex to prevent any rate limiter checks from happening while
            // the cleanup is taking place.
            mu.Lock()

            // Loop through all clients. If they haven't been seen within the last three
            // minutes, delete the corresponding entry from the map.
            for ip, client := range clients {
                if time.Since(client.lastSeen) > 3*time.Minute {
```

```

        delete(clients, ip)
    }
}

// Importantly, unlock the mutex when the cleanup is complete.
mu.Unlock()
}
}()

return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
    ip, _, err := net.SplitHostPort(r.RemoteAddr)
    if err != nil {
        app.serverErrorResponse(w, r, err)
        return
    }

    mu.Lock()

    if _, found := clients[ip]; !found {
        // Create and add a new client struct to the map if it doesn't already exist.
        clients[ip] = &client{limiter: rate.NewLimiter(2, 4)}
    }

    // Update the last seen time for the client.
    clients[ip].lastSeen = time.Now()

    if !clients[ip].limiter.Allow() {
        mu.Unlock()
        app.rateLimitExceededResponse(w, r)
        return
    }

    mu.Unlock()

    next.ServeHTTP(w, r)
})
}

```

At this point, if you restart the API and try making a batch of requests in quick succession again, you should find that the rate limiter continues to work correctly from the perspective of an individual client — just like it did before.


```
$ for i in {1..6}; do curl http://localhost:4000/v1/healthcheck; done
{
  "status": "available",
  "system_info": {
    "environment": "development",
    "version": "1.0.0"
  }
}
{
  "status": "available",
  "system_info": {
    "environment": "development",
    "version": "1.0.0"
  }
}
{
  "status": "available",
  "system_info": {
    "environment": "development",
    "version": "1.0.0"
  }
}
{
  "status": "available",
  "system_info": {
    "environment": "development",
    "version": "1.0.0"
  }
}
{
  "error": "rate limit exceeded"
}
{
  "error": "rate limit exceeded"
}
```

Additional Information

Distributed applications

Using this pattern for rate-limiting will only work if your API application is running on a single-machine. If your infrastructure is distributed, with your application running on multiple servers behind a load balancer, then you'll need to use an alternative approach.

If you're using HAProxy or Nginx as a load balancer or reverse proxy, both of these have built-in functionality for rate limiting that it would probably be sensible to use.

Alternatively, you could use a fast database like Redis to maintain a request count for clients, running on a server which all your application servers can communicate with.

Configuring the Rate Limiters

At the moment our requests-per-second and burst values are hard-coded into the `rateLimit()` middleware. This is OK, but it would be more flexible if they were configurable at runtime instead. Likewise, it would be useful to have an easy way to turn off rate limiting altogether (which is useful when you want to run benchmarks or carry out load testing, when all requests might be coming from a small number of IP addresses).

To make these things configurable, let's head back to our `cmd/api/main.go` file and update the `config` struct and command-line flags like so:

File: cmd/api/main.go

```
package main

...

type config struct {
    port int
    env string
    db struct {
        dsn string
        maxOpenConns int
        maxIdleConns int
        maxIdleTime string
    }
    // Add a new limiter struct containing fields for the requests-per-second and burst
    // values, and a boolean field which we can use to enable/disable rate limiting
    // altogether.
    limiter struct {
        rps float64
        burst int
        enabled bool
    }
}
...

func main() {
    var cfg config

    flag.IntVar(&cfg.port, "port", 4000, "API server port")
    flag.StringVar(&cfg.env, "env", "development", "Environment (development|staging|production)")

    flag.StringVar(&cfg.db.dsn, "db-dsn", os.Getenv("GREENLIGHT_DB_DSN"), "PostgreSQL DSN")

    flag.IntVar(&cfg.db.maxOpenConns, "db-max-open-conns", 25, "PostgreSQL max open connections")
    flag.IntVar(&cfg.db.maxIdleConns, "db-max-idle-conns", 25, "PostgreSQL max idle connections")
    flag.StringVar(&cfg.db.maxIdleTime, "db-max-idle-time", "15m", "PostgreSQL max connection idle time")

    // Create command line flags to read the setting values into the config struct.
    // Notice that we use true as the default for the 'enabled' setting?
    flag.Float64Var(&cfg.limiter.rps, "limiter-rps", 2, "Rate limiter maximum requests per second")
    flag.IntVar(&cfg.limiter.burst, "limiter-burst", 4, "Rate limiter maximum burst")
    flag.BoolVar(&cfg.limiter.enabled, "limiter-enabled", true, "Enable rate limiter")

    flag.Parse()

    ...
}

...
```

And then let's update our `rateLimit()` middleware to use these settings, like so:

File: cmd/api/middleware.go

```
package main

...

func (app *application) ratelimit(next http.Handler) http.Handler {

    ...

    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        // Only carry out the check if rate limiting is enabled.
        if app.config.limiter.enabled {
            ip, _, err := net.SplitHostPort(r.RemoteAddr)
            if err != nil {
                app.serverErrorResponse(w, r, err)
                return
            }

            mu.Lock()

            if _, found := clients[ip]; !found {
                clients[ip] = &client{
                    // Use the requests-per-second and burst values from the config
                    // struct.
                    limiter: rate.NewLimiter(rate.Limit(app.config.limiter.rps), app.config.limiter.burst),
                }
            }

            clients[ip].lastSeen = time.Now()

            if !clients[ip].limiter.Allow() {
                mu.Unlock()
                app.rateLimitExceededResponse(w, r)
                return
            }

            mu.Unlock()
        }

        next.ServeHTTP(w, r)
    })
}
```

Once that's done, let's try this out by running the API with the `-limiter-burst` flag and the burst value reduced to 2:

```
$ go run ./cmd/api/ -limiter-burst=2
{"level":"INFO","time":"2021-04-10T14:46:35Z","message":"database connection pool established"}
{"level":"INFO","time":"2021-04-10T14:46:35Z","message":"starting server","properties":{"addr":":4000","env":"development"}}
```

If you issue a batch of six requests in quick succession again, you should now find that only the first two succeed:

```
$ for i in {1..6}; do curl http://localhost:4000/v1/healthcheck; done
{
  "status": "available",
  "system_info": {
    "environment": "development",
    "version": "1.0.0"
  }
}
{
  "status": "available",
  "system_info": {
    "environment": "development",
    "version": "1.0.0"
  }
}
{
  "error": "rate limit exceeded"
}
{
  "error": "rate limit exceeded"
}
{
  "error": "rate limit exceeded"
}
{
  "error": "rate limit exceeded"
}
```

Similarly, you can try disabling the rate limiter altogether with the `-limiter-enabled=false` flag like so:

```
$ go run ./cmd/api/ -limiter-enabled=false
{"level":"INFO","time":"2021-04-10T14:53:33Z","message":"database connection pool established"}
{"level":"INFO","time":"2021-04-10T14:53:33Z","message":"starting server","properties":{"addr":":4000","env":"development"}}
```

And you should find that all requests now complete successfully, no matter how many you make.

```
$ for i in {1..6}; do curl http://localhost:4000/v1/healthcheck; done
{
  "status": "available",
  "system_info": {
    "environment": "development",
    "version": "1.0.0"
  }
}
...
{
  "status": "available",
  "system_info": {
    "environment": "development",
    "version": "1.0.0"
  }
}
```

Graceful Shutdown

In this next section of the book we're going to talk about an important but often overlooked topic: *how to safely stop your running application*.

At the moment, when we stop our API application (usually by pressing **Ctrl+C**) it is terminated *immediately* with no opportunity for in-flight HTTP requests to complete. This isn't ideal for two reasons:

- It means that clients won't receive responses to their in-flight requests — all they will experience is a hard closure of the HTTP connection.
- Any work being carried out by our handlers may be left in an incomplete state.

We're going to mitigate these problems by adding *graceful shutdown* functionality to our application, so that in-flight HTTP requests have the opportunity to finish being processed *before* the application is terminated.

You'll learn about:

- Shutdown signals — what they are, how to send them, and how to listen for them in your API application.
- How to use these signals to trigger a graceful shutdown of the HTTP server using Go's `Shutdown()` method.

Sending Shutdown Signals

When our application is running, we can terminate it at any time by sending it a specific [signal](#). A common way to do this, which you've probably been using, is by pressing **Ctrl+C** on your keyboard to send an *interrupt* signal — also known as a **SIGINT**.

But this isn't the only type of signal which can stop our application. Some of the other common ones are:

Signal	Description	Keyboard shortcut	Catchable
SIGINT	Interrupt from keyboard	Ctrl+C	Yes
SIGQUIT	Quit from keyboard	Ctrl+\	Yes
SIGKILL	Kill process (terminate immediately)	-	No
SIGTERM	Terminate process in orderly manner	-	Yes

It's important to explain upfront that some signals are *catchable* and others are not. Catchable signals can be intercepted by our application and either ignored, or used to trigger a certain action (such as a graceful shutdown). Other signals, like **SIGKILL**, are not catchable and cannot be intercepted.

Let's take a quick look at these signals in action. If you're following along, go ahead and start the API application with the same **go run ./cmd/api** command as normal:

```
$ go run ./cmd/api
{"level":"INFO","time":"2021-04-11T07:33:39Z","message":"database connection pool established"}
{"level":"INFO","time":"2021-04-11T07:33:39Z","message":"starting server","properties":{"addr":":4000","env":"development"}}
```

Doing this should start a process with the name **api** on your machine. You can use the **pgrep** command to verify that this process exists, like so:

```
$ pgrep -l api
4414 api
```

In my case, I can see that the **api** process is running and has the *process ID* **4414** (your process ID will most likely be different).

Once that's confirmed, go ahead and try sending a **SIGKILL** signal to the **api** process using the **kill** command like so:

```
$ kill -SIGKILL api
```

If you go back to the terminal window that is running the API application, you should see that it has been terminated and the final line in the output stream is **signal: killed**. Similar to this:

```
$ go run ./cmd/api
{"level":"INFO","time":"2021-04-11T07:33:39Z","message":"database connection pool established"}
{"level":"INFO","time":"2021-04-11T07:33:39Z","message":"starting server","properties":{"addr":":4000","env":"development"}}
signal: killed
```

Feel free to repeat the same process, but sending a **SIGTERM** signal instead:

```
$ kill -SIGTERM api
```

This time you should see the line **signal: terminated** at the end of the output, like so:

```
$ go run ./cmd/api
{"level":"INFO","time":"2021-04-11T07:35:54Z","message":"database connection pool established"}
{"level":"INFO","time":"2021-04-11T07:35:54Z","message":"starting server","properties":{"addr":":4000","env":"development"}}
signal: terminated
```

You can also try sending a **SIGQUIT** signal — either by pressing **Ctrl+** on your keyboard or running **kill -SIGQUIT api**. This will cause the application to exit with a stack dump, similar to this:


```

$ go run ./cmd/api
{"level":"INFO","time":"2021-04-11T07:40:37Z","message":"database connection pool established"}
{"level":"INFO","time":"2021-04-11T07:40:37Z","message":"starting server","properties":{"addr":":4000","env":"development"}}
SIGQUIT: quit
PC=0x46ebe1 m=0 sigcode=0

goroutine 0 [idle]:
runtime.futex(0x964870, 0x80, 0x0, 0x0, 0x0, 0x964720, 0x7ffd551034f8, 0x964420, 0x7ffd55103508, 0x40dcbf, ...)
    /usr/local/go/src/runtime/sys_linux_amd64.s:579 +0x21
runtime.futexsleep(0x964870, 0x0, 0xffffffffffffffff)
    /usr/local/go/src/runtime/os_linux.go:44 +0x46
runtime.notesleep(0x964870)
    /usr/local/go/src/runtime/lock_futex.go:159 +0x9f
runtime.mPark()
    /usr/local/go/src/runtime/proc.go:1340 +0x39
runtime.stopm()
    /usr/local/go/src/runtime/proc.go:2257 +0x92
runtime.findrunnable(0xc00002c000, 0x0)
    /usr/local/go/src/runtime/proc.go:2916 +0x72e
runtime.schedule()
    /usr/local/go/src/runtime/proc.go:3125 +0x2d7
runtime.park_m(0xc000000180)
    /usr/local/go/src/runtime/proc.go:3274 +0x9d
runtime.mcall(0x0)
    /usr/local/go/src/runtime/asm_amd64.s:327 +0x5b
...

```

We can see that these signals are effective in terminating our application — but the problem we have is that they all cause our application to exit *immediately*.

Fortunately, Go provides tools in the [os/signal](#) package that we can use to intercept catchable signals and trigger a graceful shutdown of our application. We'll look at how to do that in the next chapter.

Intercepting Shutdown Signals

Before we get into the nuts and bolts of how to intercept signals, let's move the code related to our `http.Server` out of the `main()` function and into a separate file. This will give us a clean and clear starting point from which we can build up the graceful shutdown functionality.

If you're following along, create a new `cmd/api/server.go` file:

```
$ touch cmd/api/server.go
```

And then add a new `app.serve()` method which initializes and starts our `http.Server`, like so:

```
File: cmd/api/server.go

package main

import (
    "fmt"
    "net/http"
    "time"
)

func (app *application) serve() error {
    // Declare a HTTP server using the same settings as in our main() function.
    srv := &http.Server{
        Addr:         fmt.Sprintf(":%d", app.config.port),
        Handler:      app.routes(),
        IdleTimeout:  time.Minute,
        ReadTimeout:  10 * time.Second,
        WriteTimeout: 30 * time.Second,
    }

    // Likewise log a "starting server" message.
    app.logger.Printf("starting server", map[string]string{
        "addr": srv.Addr,
        "env":  app.config.env,
    })

    // Start the server as normal, returning any error.
    return srv.ListenAndServe()
}
```

With that in place, we can simplify our `main()` function to use this new `app.serve()` method like so:

File: cmd/api/main.go

```
package main

import (
    "context"
    "database/sql"
    "flag"
    "os"
    "time"

    "greenlight.alexedwards.net/internal/data"
    "greenlight.alexedwards.net/internal/jsonlog"

    _ "github.com/lib/pq"
)

...

func main() {
    var cfg config

    flag.IntVar(&cfg.port, "port", 4000, "API server port")
    flag.StringVar(&cfg.env, "env", "development", "Environment (development|staging|production)")

    flag.StringVar(&cfg.db.dsn, "db-dsn", os.Getenv("GREENLIGHT_DB_DSN"), "PostgreSQL DSN")

    flag.IntVar(&cfg.db.maxOpenConns, "db-max-open-conns", 25, "PostgreSQL max open connections")
    flag.IntVar(&cfg.db.maxIdleConns, "db-max-idle-conns", 25, "PostgreSQL max idle connections")
    flag.StringVar(&cfg.db.maxIdleTime, "db-max-idle-time", "15m", "PostgreSQL max connection idle time")

    flag.Float64Var(&cfg.limiter.rps, "limiter-rps", 2, "Rate limiter maximum requests per second")
    flag.IntVar(&cfg.limiter.burst, "limiter-burst", 4, "Rate limiter maximum burst")
    flag.BoolVar(&cfg.limiter.enabled, "limiter-enabled", true, "Enable rate limiter")

    flag.Parse()

    logger := jsonlog.New(os.Stdout, jsonlog.LevelInfo)

    db, err := openDB(cfg)
    if err != nil {
        logger.Fatalf(err, nil)
    }
    defer db.Close()

    logger.Printf("database connection pool established", nil)

    app := &application{
        config: cfg,
        logger: logger,
        models: data.NewModels(db),
    }

    // Call app.serve() to start the server.
    err = app.serve()
    if err != nil {
        logger.Fatalf(err, nil)
    }
}

...
```

Catching SIGINT and SIGTERM signals

The next thing that we want to do is update our application so that it ‘catches’ any **SIGINT** and **SIGTERM** signals. As we mentioned above, **SIGKILL** signals are not catchable (and will always cause the application to terminate immediately), and we’ll leave **SIGQUIT** with its default behavior (as it’s handy if you want to execute a non-graceful shutdown via a keyboard shortcut).

To catch the signals, we’ll need to spin up a background goroutine which runs for the lifetime of our application. In this background goroutine, we can use the `signal.Notify()` function to listen for specific signals and relay them to a channel for further processing.

I’ll demonstrate.

Open up the `cmd/api/server.go` file and update it like so:

File: cmd/api/server.go

```
package main

import (
    "fmt"
    "net/http"
    "os"           // New import
    "os/signal"    // New import
    "syscall"      // New import
    "time"
)

func (app *application) serve() error {
    srv := &http.Server{
        Addr:         fmt.Sprintf(":%d", app.config.port),
        Handler:      app.routes(),
        IdleTimeout:  time.Minute,
        ReadTimeout:  10 * time.Second,
        WriteTimeout: 30 * time.Second,
    }

    // Start a background goroutine.
    go func() {
        // Create a quit channel which carries os.Signal values.
        quit := make(chan os.Signal, 1)

        // Use signal.Notify() to listen for incoming SIGINT and SIGTERM signals and
        // relay them to the quit channel. Any other signals will not be caught by
        // signal.Notify() and will retain their default behavior.
        signal.Notify(quit, syscall.SIGINT, syscall.SIGTERM)

        // Read the signal from the quit channel. This code will block until a signal is
        // received.
        s := <-quit

        // Log a message to say that the signal has been caught. Notice that we also
        // call the String() method on the signal to get the signal name and include it
        // in the log entry properties.
        app.logger.Printf("caught signal", map[string]string{
            "signal": s.String(),
        })

        // Exit the application with a 0 (success) status code.
        os.Exit(0)
    }()

    // Start the server as normal.
    app.logger.Printf("starting server", map[string]string{
        "addr": srv.Addr,
        "env":  app.config.env,
    })

    return srv.ListenAndServe()
}
```

At the moment this new code isn't doing much — after intercepting the signal, all we do is log a message and then exit our application. But the important thing is that it demonstrates the pattern of how to catch specific signals and handle them in your code.

One thing I'd like to quickly emphasize about this: our `quit` channel is a [buffered channel](#)

with size 1.

We need to use a buffered channel here because `signal.Notify()` does not wait for a receiver to be available when sending a signal to the `quit` channel. If we had used a regular (non-buffered) channel here instead, a signal could be ‘missed’ if our `quit` channel is not ready to receive at the exact moment that the signal is sent. By using a buffered channel, we avoid this problem and ensure that we never miss a signal.

OK, let’s try this out.

First, run the application and then press `Ctrl+C` on your keyboard to send a `SIGINT` signal. You should see a "caught signal" log entry with `"signal":"interrupt"` in the properties, similar to this:

```
$ go run ./cmd/api
{"level":"INFO","time":"2021-04-11T10:42:28Z","message":"database connection pool established"}
{"level":"INFO","time":"2021-04-11T10:42:28Z","message":"starting server","properties":{"addr":":4000","env":"development"}}
^C{"level":"INFO","time":"2021-04-11T10:42:29Z","message":"caught signal","properties":{"signal":"interrupt"}}
```

You can also restart the application and try sending a `SIGTERM` signal. This time the log entry properties should contain `"signal":"terminated"`, like so:

```
$ pkill -SIGTERM api
```

```
$ go run ./cmd/api
{"level":"INFO","time":"2021-04-11T10:42:46Z","message":"database connection pool established"}
{"level":"INFO","time":"2021-04-11T10:42:46Z","message":"starting server","properties":{"addr":":4000","env":"development"}}
{"level":"INFO","time":"2021-04-11T10:42:51Z","message":"caught signal","properties":{"signal":"terminated"}}
```

In contrast, sending a `SIGKILL` or `SIGQUIT` signal will continue to cause the application to exit immediately *without the signal being caught*, so you shouldn’t see a "caught signal" message in the logs. For example, if you restart the application and issue a `SIGKILL`...

```
$ pkill -SIGKILL api
```

The application should be terminated immediately, and the logs will look like this:

```
$ go run ./cmd/api
{"level":"INFO","time":"2021-04-11T10:43:12Z","message":"database connection pool established"}
{"level":"INFO","time":"2021-04-11T10:43:12Z","message":"starting server","properties":{"addr":":4000","env":"development"}}
signal: killed
```

Executing the Shutdown

Intercepting the signals is all well and good, but it's not very useful until we do something with them! In this chapter, we're going to update our application so that the **SIGINT** and **SIGTERM** signals we intercept trigger a graceful shutdown of our API.

Specifically, after receiving one of these signals we will call the `Shutdown()` method on our HTTP server. The official documentation describes this as follows:

Shutdown gracefully shuts down the server without interrupting any active connections. Shutdown works by first closing all open listeners, then closing all idle connections, and then waiting indefinitely for connections to return to idle and then shut down.

The pattern to implement this in practice is difficult to describe with words, so let's jump into the code and talk through the details as we go along.

File: cmd/api/server.go

```
package main

import (
    "context" // New import
    "errors"  // New import
    "fmt"
    "net/http"
    "os"
    "os/signal"
    "syscall"
    "time"
)

func (app *application) serve() error {
    srv := &http.Server{
        Addr:      fmt.Sprintf(":%d", app.config.port),
        Handler:   app.routes(),
        IdleTimeout: time.Minute,
        ReadTimeout: 10 * time.Second,
        WriteTimeout: 30 * time.Second,
    }

    // Create a shutdownError channel. We will use this to receive any errors returned
    // by the graceful Shutdown() function.
    shutdownError := make(chan error)

    go func() {
        // Intercept the signals, as before.
        quit := make(chan os.Signal, 1)
        signal.Notify(quit, syscall.SIGINT, syscall.SIGTERM)
        s := <-quit

        // Update the log entry to say "shutting down server" instead of "caught signal".
        app.logger.Printf("shutting down server", map[string]string{
```

```

        "signal": s.String(),
    })

    // Create a context with a 5-second timeout.
    ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
    defer cancel()

    // Call Shutdown() on our server, passing in the context we just made.
    // Shutdown() will return nil if the graceful shutdown was successful, or an
    // error (which may happen because of a problem closing the listeners, or
    // because the shutdown didn't complete before the 5-second context deadline is
    // hit). We relay this return value to the shutdownError channel.
    shutdownError <- srv.Shutdown(ctx)
}()

app.logger.Println("starting server", map[string]string{
    "addr": srv.Addr,
    "env": app.config.env,
})

// Calling Shutdown() on our server will cause ListenAndServe() to immediately
// return a http.ErrServerClosed error. So if we see this error, it is actually a
// good thing and an indication that the graceful shutdown has started. So we check
// specifically for this, only returning the error if it is NOT http.ErrServerClosed.
err := srv.ListenAndServe()
if !errors.Is(err, http.ErrServerClosed) {
    return err
}

// Otherwise, we wait to receive the return value from Shutdown() on the
// shutdownError channel. If return value is an error, we know that there was a
// problem with the graceful shutdown and we return the error.
err = <-shutdownError
if err != nil {
    return err
}

// At this point we know that the graceful shutdown completed successfully and we
// log a "stopped server" message.
app.logger.Println("stopped server", map[string]string{
    "addr": srv.Addr,
})

return nil
}

```

At first glance this code might seem a bit complex, but at a high-level what it's doing can be summarized very simply: *when we receive a **SIGINT** or **SIGTERM** signal, we instruct our server to stop accepting any new HTTP requests, and give any in-flight requests a 'grace period' of 5 seconds to complete before the application is terminated.*

But there are a couple of important details to be aware of:

- The **Shutdown()** method does not wait for any background tasks to complete, nor does it close hijacked long-lived connections like WebSockets. Instead, you will need to implement your own logic to coordinate a graceful shutdown of these things. We'll look at some techniques for doing this later in the book.

- There is an [open bug](#) which effectively limits the grace period to 5 seconds for HTTP/2 connections. HTTP/2 is automatically supported for HTTPS connections in Go, so if you're planning to handle HTTPS connections from your Go application then (for now) it is sensible to use a sub 5-second grace period for your context timeout. If it wasn't for this bug, I would suggest using a longer grace period in this chapter (around 20 seconds).

But those things aside, this should now be working nicely in our application.

To help demonstrate the graceful shutdown functionality, you can add a 4 second sleep delay to the `healthcheckHandler` method, like so:

```
File: cmd/api/healthcheck.go

package main

import (
    "net/http"
    "time" // New import
)

func (app *application) healthcheckHandler(w http.ResponseWriter, r *http.Request) {
    env := envelope{
        "status": "available",
        "system_info": map[string]string{
            "environment": app.config.env,
            "version":     version,
        },
    }

    // Add a 4 second delay.
    time.Sleep(4 * time.Second)

    err := app.writeJSON(w, http.StatusOK, env, nil)
    if err != nil {
        app.serverErrorResponse(w, r, err)
    }
}
```

Then start the API, and in another terminal window issue a request to the healthcheck endpoint followed by a `SIGTERM` signal.

```
$ curl localhost:4000/v1/healthcheck & kill -SIGTERM api
```

In the logs for the server, you should immediately see a `shutting down server` message following the `SIGTERM` signal, similar to this:

```
$ go run ./cmd/api
{"level":"INFO","time":"2021-04-11T10:54:55Z","message":"database connection pool established"}
{"level":"INFO","time":"2021-04-11T10:54:55Z","message":"starting server","properties":{"addr":":4000","env":"development"}}
{"level":"INFO","time":"2021-04-11T10:55:03Z","message":"shutting down server","properties":{"signal":"terminated"}}
```

Then after a 4 second delay for the in-flight request to complete, our `healthcheckHandler` should return the JSON response as normal and you should see that our API has logged a final `"stopped server"` message before exiting cleanly:

```
$ go run ./cmd/api
{"level":"INFO","time":"2021-04-11T10:54:55Z","message":"database connection pool established"}
{"level":"INFO","time":"2021-04-11T10:54:55Z","message":"starting server","properties":{"addr":":4000","env":"development"}}
{"level":"INFO","time":"2021-04-11T10:55:03Z","message":"shutting down server","properties":{"signal":"terminated"}}
{"level":"INFO","time":"2021-04-11T10:55:07Z","message":"stopped server","properties":{"addr":":4000"}}
```

Notice the 4 second delay between the `"shutting down server"` and `"stopped server"` messages in the timestamps above?

So this is working really well now. Any time that we want to gracefully shutdown our application, we can do so by sending a `SIGINT` (`Ctrl+C`) or `SIGTERM` signal. So long as no in-flight requests take more than 5 seconds to complete, our handlers will have time to complete their work and our clients will receive a proper HTTP response. And if we ever want to exit immediately, without a graceful shutdown, we can still do so by sending a `SIGQUIT` (`Ctrl+\`) or `SIGKILL` signal instead.

Lastly, if you're following along, please revert `healthcheckHandler` to remove the 4 second sleep. Like so:

```
File: cmd/api/healthcheck.go

package main

import (
    "net/http"
)

func (app *application) healthcheckHandler(w http.ResponseWriter, r *http.Request) {
    env := envelope{
        "status": "available",
        "system_info": map[string]string{
            "environment": app.config.env,
            "version":     version,
        },
    },
}

err := app.writeJSON(w, http.StatusOK, env, nil)
if err != nil {
    app.serverErrorResponse(w, r, err)
}
}
```

User Model Setup and Registration

In the upcoming sections of this book, we're going to shift our focus towards *users*: registering them, activating them, authenticating them, and restricting access to our API endpoints depending on the permissions that they have.

But before we can do these things, we need to lay some groundwork. Specifically we need to:

- Create a new `users` table in PostgreSQL for storing our user data.
- Create a `UserModel` which contains the code for interacting with our `users` table, validating user data, and hashing user passwords.
- Develop a `POST /v1/users` endpoint which can be used to register new users in our application.

Setting up the Users Database Table

Let's begin by creating a new **users** table in our database. If you're following along, use the **migrate** tool to generate a new pair of SQL migration files:

```
$ migrate create -seq -ext=.sql -dir=./migrations create_users_table
/home/alex/Projects/greenlight/migrations/000004_create_users_table.up.sql
/home/alex/Projects/greenlight/migrations/000004_create_users_table.down.sql
```

And then add the following SQL statements to the 'up' and 'down' files respectively:

File: migrations/000004_create_users_table.up.sql

```
CREATE TABLE IF NOT EXISTS users (
  id bigserial PRIMARY KEY,
  created_at timestamp(0) with time zone NOT NULL DEFAULT NOW(),
  name text NOT NULL,
  email citext UNIQUE NOT NULL,
  password_hash bytea NOT NULL,
  activated bool NOT NULL,
  version integer NOT NULL DEFAULT 1
);
```

File: migrations/000004_create_users_table.down.sql

```
DROP TABLE IF EXISTS users;
```

There are a few interesting about this **CREATE TABLE** statement that I'd like to quickly explain:

1. The **email** column has the type **citext** (case-insensitive text). This type stores text data exactly as it is inputted — without changing the case in any way — but *comparisons* against the data are always case-insensitive... including lookups on associated indexes.
2. We've also got a **UNIQUE** constraint on the **email** column. Combined with the **citext** type, this means that no two rows in the database can have the same email value — even if they have different cases. This essentially enforces a database-level business rule that *no two users should exist with the same email address*.
3. The **password_hash** column has the type **bytea** (binary string). In this column we'll store a *one-way hash of the user's password* generated using **bcrypt** — not the plaintext

password itself.

4. The **activated** column stores a boolean value to denote whether a user account is 'active' or not. We will set this to **false** by default when creating a new user, and require the user to confirm their email address before we set it to **true**.
5. We've also included a **version** number column, which we will increment each time a user record is updated. This will allow us to use optimistic locking to prevent race conditions when updating user records, in the same way that we did with movies earlier in the book.

OK, let's execute the 'up' migration:

```
$ migrate -path=./migrations -database=$GREENLIGHT_DB_DSN up
4/u create_users_table (62.43511ms)
```

And then you should be able to connect to your database and verify that the new **users** table has been created, as expected:

```
$ psql $GREENLIGHT_DB_DSN
psql (13.2 (Ubuntu 13.2-1.pgdg20.04+1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression: off)
Type "help" for help.

greenlight=> \d users

              Table "public.users"
   Column   |      Type      | Collation | Nullable |   Default   |
-----+-----+-----+-----+-----+
 id          | bigint          |           | not null | nextval('users_id_seq'::regclass)
 created_at  | timestamp(0) with time zone |           | not null | now()
 name        | text            |           | not null |
 email       | citext          |           | not null |
 password_hash | bytea          |           | not null |
 activated   | boolean         |           | not null |
 version     | integer         |           | not null | 1
Indexes:
    "users_pkey" PRIMARY KEY, btree (id)
    "users_email_key" UNIQUE CONSTRAINT, btree (email)
```

One important thing to point out here: the **UNIQUE** constraint on our **email** column has automatically been assigned the name **users_email_key**. This will become relevant in the next chapter, when we need to handle any errors caused by a user registering twice with the same email address.

Setting up the Users Model

Now that our database table is set up, we're going to update our `internal/data` package to contain a new `User` struct (to represent the data for an individual user), and create a `UserModel` type (which we will use to perform various SQL queries against our `users` table).

If you're following along, go ahead and create an `internal/data/users.go` file to hold this new code:

```
$ touch internal/data/users.go
```

Let's start by defining the `User` struct, along with some helper methods for setting and verifying the password for a user.

As we mentioned earlier, in this project we will use `bcrypt` to hash user passwords before storing them in the database. So the first thing we need to do is install the golang.org/x/crypto/bcrypt package, which provides an easy-to-use Go implementation of the `bcrypt` algorithm.

```
$ go get golang.org/x/crypto/bcrypt@latest
go: downloading golang.org/x/crypto v0.0.0-20210817164053-32db794688a5
go get: added golang.org/x/crypto v0.0.0-20210817164053-32db794688a5
```

Then in the `internal/data/users.go` file, go ahead and create the `User` struct and helper methods like so:

File: internal/data/users.go

```
package data

import (
    "errors"
    "time"

    "golang.org/x/crypto/bcrypt"
)

// Define a User struct to represent an individual user. Importantly, notice how we are
// using the json:"- " struct tag to prevent the Password and Version fields appearing in
// any output when we encode it to JSON. Also notice that the Password field uses the
// custom password type defined below.
type User struct {
    ID          int64    `json:"id"`
    CreatedAt   time.Time `json:"created_at"`
    Name        string    `json:"name"`
    Email       string    `json:"email"`
    Password     password  `json:"- "`
    Activated   bool      `json:"activated"`
    Version     int       `json:"- "`
}

// Create a custom password type which is a struct containing the plaintext and hashed
// versions of the password for a user. The plaintext field is a *pointer* to a string,
// so that we're able to distinguish between a plaintext password not being present in
// the struct at all, versus a plaintext password which is the empty string "".
type password struct {
    plaintext *string
    hash      []byte
}

// The Set() method calculates the bcrypt hash of a plaintext password, and stores both
// the hash and the plaintext versions in the struct.
func (p *password) Set(plaintextPassword string) error {
    hash, err := bcrypt.GenerateFromPassword([]byte(plaintextPassword), 12)
    if err != nil {
        return err
    }

    p.plaintext = &plaintextPassword
    p.hash = hash

    return nil
}

// The Matches() method checks whether the provided plaintext password matches the
// hashed password stored in the struct, returning true if it matches and false
// otherwise.
func (p *password) Matches(plaintextPassword string) (bool, error) {
    err := bcrypt.CompareHashAndPassword(p.hash, []byte(plaintextPassword))
    if err != nil {
        switch {
        case errors.Is(err, bcrypt.ErrMismatchedHashAndPassword):
            return false, nil
        default:
            return false, err
        }
    }

    return true, nil
}
```

We explained how the golang.org/x/crypto/bcrypt package works previously in *Let's Go*, but let's quickly recap the key points:

- The `bcrypt.GenerateFromPassword()` function generates a bcrypt hash of a password using a specific cost parameter (in the code above, we use a cost of `12`). The higher the cost, the slower and more computationally expensive it is to generate the hash. There is a balance to be struck here — we want the cost to be prohibitively expensive for attackers, but also not so *slow* that it harms the user experience of our API. This function returns a *hash string* in the format:

```
$2b$[cost]$[22-character salt][31-character hash]
```

- The `bcrypt.CompareHashAndPassword()` function works by re-hashing the provided password using the *same salt and cost parameter* that is in the hash string that we're comparing against. The re-hashed value is then checked against the original hash string using the `subtle.ConstantTimeCompare()` function, which performs a comparison in constant time (to mitigate the risk of a timing attack). If they don't match, then it will return a `bcrypt.ErrMismatchedHashAndPassword` error.

Adding Validation Checks

Let's move on and create some validation checks for our `User` struct. Specifically, we want to:

- Check that the `Name` field is not the empty string, and the value is less than 500 bytes long.
- Check that the `Email` field is not the empty string, and that it matches the regular expression for email addresses that we added in our `validator` package earlier in the book.
- If the `Password.plaintext` field is not `nil`, then check that the value is not the empty string and is between 8 and 72 bytes long.
- Check that the `Password.hash` field is never `nil`.

Note: When creating a bcrypt hash the input is [truncated to a maximum of 72 bytes](#). So, if someone uses a very long password, it means that any bytes after that would effectively be ignored when creating the hash. To avoid any confusion for users, we'll simply enforce a hard maximum length of 72 bytes on the password in our validation checks. If you don't want to enforce a maximum length, you could [pre-hash the password](#) instead.

Additionally, we're going to want to use the email and plaintext password validation checks again independently later in the book, so we'll define those checks in some standalone functions.

Go ahead and update the [internal/data/users.go](#) file like so:

File: internal/data/users.go

```
package data

import (
    "errors"
    "time"

    "greenlight.alexedwards.net/internal/validator" // New import

    "golang.org/x/crypto/bcrypt"
)

...

func ValidateEmail(v *validator.Validator, email string) {
    v.Check(email != "", "email", "must be provided")
    v.Check(validator.Matches(email, validator.EmailRX), "email", "must be a valid email address")
}

func ValidatePasswordPlaintext(v *validator.Validator, password string) {
    v.Check(password != "", "password", "must be provided")
    v.Check(len(password) >= 8, "password", "must be at least 8 bytes long")
    v.Check(len(password) <= 72, "password", "must not be more than 72 bytes long")
}

func ValidateUser(v *validator.Validator, user *User) {
    v.Check(user.Name != "", "name", "must be provided")
    v.Check(len(user.Name) <= 500, "name", "must not be more than 500 bytes long")

    // Call the standalone ValidateEmail() helper.
    ValidateEmail(v, user.Email)

    // If the plaintext password is not nil, call the standalone
    // ValidatePasswordPlaintext() helper.
    if user.Password.plaintext != nil {
        ValidatePasswordPlaintext(v, *user.Password.plaintext)
    }

    // If the password hash is ever nil, this will be due to a logic error in our
    // codebase (probably because we forgot to set a password for the user). It's a
    // useful sanity check to include here, but it's not a problem with the data
    // provided by the client. So rather than adding an error to the validation map we
    // raise a panic instead.
    if user.Password.hash == nil {
        panic("missing password hash for user")
    }
}
```

Creating the UserModel

The next step in this process is setting up a `UserModel` type which isolates the database interactions with our PostgreSQL `users` table.

We'll follow the same pattern here that we used for our `MovieModel`, and implement the following three methods:

- `Insert()` to create a new user record in the database.

- `GetByEmail()` to retrieve the data for a user with a specific email address.
- `Update()` to change the data for a specific user.

Open up the `internal/data/users.go` file again, and add the following code:

File: internal/data/users.go

```
package data

import (
    "context"          // New import
    "database/sql"      // New import
    "errors"
    "time"

    "greenlight.alexedwards.net/internal/validator"

    "golang.org/x/crypto/bcrypt"
)

// Define a custom ErrDuplicateEmail error.
var (
    ErrDuplicateEmail = errors.New("duplicate email")
)

...

// Create a UserModel struct which wraps the connection pool.
type UserModel struct {
    DB *sql.DB
}

// Insert a new record in the database for the user. Note that the id, created_at and
// version fields are all automatically generated by our database, so we use the
// RETURNING clause to read them into the User struct after the insert, in the same way
// that we did when creating a movie.
func (m UserModel) Insert(user *User) error {
    query := `
        INSERT INTO users (name, email, password_hash, activated)
        VALUES ($1, $2, $3, $4)
        RETURNING id, created_at, version`

    args := []interface{}{user.Name, user.Email, user.Password.hash, user.Activated}

    ctx, cancel := context.WithTimeout(context.Background(), 3*time.Second)
    defer cancel()

    // If the table already contains a record with this email address, then when we try
    // to perform the insert there will be a violation of the UNIQUE "users_email_key"
    // constraint that we set up in the previous chapter. We check for this error
    // specifically, and return custom ErrDuplicateEmail error instead.
    err := m.DB.QueryRowContext(ctx, query, args...).Scan(&user.ID, &user.CreatedAt, &user.Version)
    if err != nil {
        switch {
            case err.Error() == `pq: duplicate key value violates unique constraint "users_email_key"`:
                return ErrDuplicateEmail
            default:
                return err
        }
    }
}

return nil
```

```

}

// Retrieve the User details from the database based on the user's email address.
// Because we have a UNIQUE constraint on the email column, this SQL query will only
// return one record (or none at all, in which case we return a ErrRecordNotFound error).
func (m UserModel) GetByEmail(email string) (*User, error) {
    query := `
        SELECT id, created_at, name, email, password_hash, activated, version
        FROM users
        WHERE email = $1`

    var user User

    ctx, cancel := context.WithTimeout(context.Background(), 3*time.Second)
    defer cancel()

    err := m.DB.QueryRowContext(ctx, query, email).Scan(
        &user.ID,
        &user.CreatedAt,
        &user.Name,
        &user.Email,
        &user.Password.hash,
        &user.Activated,
        &user.Version,
    )

    if err != nil {
        switch {
        case errors.Is(err, sql.ErrNoRows):
            return nil, ErrRecordNotFound
        default:
            return nil, err
        }
    }

    return &user, nil
}

// Update the details for a specific user. Notice that we check against the version
// field to help prevent any race conditions during the request cycle, just like we did
// when updating a movie. And we also check for a violation of the "users_email_key"
// constraint when performing the update, just like we did when inserting the user
// record originally.
func (m UserModel) Update(user *User) error {
    query := `
        UPDATE users
        SET name = $1, email = $2, password_hash = $3, activated = $4, version = version + 1
        WHERE id = $5 AND version = $6
        RETURNING version`

    args := []interface{}{
        user.Name,
        user.Email,
        user.Password.hash,
        user.Activated,
        user.ID,
        user.Version,
    }

    ctx, cancel := context.WithTimeout(context.Background(), 3*time.Second)
    defer cancel()

    err := m.DB.QueryRowContext(ctx, query, args...).Scan(&user.Version)
    if err != nil {
        switch {
        case err.Error() == `pq: duplicate key value violates unique constraint "users_email_key"`:
            return ErrEmailAlreadyExists
        default:
            return err
        }
    }

    return nil
}

```

```

        return ErrDuplicateEmail
    case errors.Is(err, sql.ErrNoRows):
        return ErrEditConflict
    default:
        return err
    }
}

return nil
}

```

Hopefully that feels nice and straightforward — we’re using the same code patterns that we did for the CRUD operations on our `movies` table earlier in the book.

The only difference is that in some of the methods we’re specifically checking for any errors due to a violation of our unique `users_email_key` constraint. As we’ll see in the next chapter, by treating this as a special case we’ll be able to respond to clients with a message to say that “*this email address is already in use*”, rather than sending them a `500 Internal Server Error` response like we normally would.

To finish all this off, the final thing we need to do is update our `internal/data/models.go` file to include the new `UserModel` in our parent `Models` struct. Like so:

```

File: internal/data/models.go

package data

...

type Models struct {
    Movies MovieModel
    Users  UserModel // Add a new Users field.
}

func NewModels(db *sql.DB) Models {
    return Models{
        Movies: MovieModel{DB: db},
        Users:  UserModel{DB: db}, // Initialize a new UserModel instance.
    }
}

```

Registering a User

Now that we've laid the groundwork, let's start putting it to use by creating a new API endpoint to manage the process of registering (or *signing up*) a new user.

Method	URL Pattern	Handler	Action
GET	/v1/healthcheck	healthcheckHandler	Show application information
GET	/v1/movies	listMoviesHandler	Show the details of all movies
POST	/v1/movies	createMovieHandler	Create a new movie
GET	/v1/movies/:id	showMovieHandler	Show the details of a specific movie
PATCH	/v1/movies/:id	updateMovieHandler	Update the details of a specific movie
DELETE	/v1/movies/:id	deleteMovieHandler	Delete a specific movie
POST	/v1/users	registerUserHandler	Register a new user

When a client calls this new **POST /v1/users** endpoint, we will expect them to provide the following details for the new user in a JSON request body. Similar to this:

```
{
  "name": "Alice Smith",
  "email": "alice@example.com",
  "password": "pa55word"
}
```

When we receive this, the **registerUserHandler** should create a new **User** struct containing these details, validate it with the **ValidateUser()** helper, and then pass it to our **UserModel.Insert()** method to create a new database record.

In fact, we've already written most of the code we need for the **registerUserHandler** — it's now just a case of piecing it all together in the correct order.

If you're following along, go ahead and create a new **cmd/api/users.go** file:

```
$ touch cmd/api/users.go
```

And then add the new `registerUserHandler` method containing the following code:

File: cmd/api/users.go

```
package main

import (
    "errors"
    "net/http"

    "greenlight.alexedwards.net/internal/data"
    "greenlight.alexedwards.net/internal/validator"
)

func (app *application) registerUserHandler(w http.ResponseWriter, r *http.Request) {
    // Create an anonymous struct to hold the expected data from the request body.
    var input struct {
        Name      string `json:"name"`
        Email     string `json:"email"`
        Password  string `json:"password"`
    }

    // Parse the request body into the anonymous struct.
    err := app.readJSON(w, r, &input)
    if err != nil {
        app.badRequestResponse(w, r, err)
        return
    }

    // Copy the data from the request body into a new User struct. Notice also that we
    // set the Activated field to false, which isn't strictly necessary because the
    // Activated field will have the zero-value of false by default. But setting this
    // explicitly helps to make our intentions clear to anyone reading the code.
    user := &data.User{
        Name:      input.Name,
        Email:     input.Email,
        Activated: false,
    }

    // Use the Password.Set() method to generate and store the hashed and plaintext
    // passwords.
    err = user.Password.Set(input.Password)
    if err != nil {
        app.serverErrorResponse(w, r, err)
        return
    }

    v := validator.New()

    // Validate the user struct and return the error messages to the client if any of
    // the checks fail.
    if data.ValidateUser(v, user); !v.Valid() {
        app.failedValidationResponse(w, r, v.Errors)
        return
    }

    // Insert the user data into the database.
    err = app.models.Users.Insert(user)
    if err != nil {
        switch {
            // If we get a ErrDuplicateEmail error, use the v.AddError() method to manually
            // add a message to the validator instance, and then call our
            // failedValidationResponse() helper.
            case errors.Is(err, data.ErrDuplicateEmail):
```

```

        v.AddError("email", "a user with this email address already exists")
        app.failedValidationResponse(w, r, v.Errors)
    default:
        app.serverErrorResponse(w, r, err)
    }
    return
}

// Write a JSON response containing the user data along with a 201 Created status
// code.
err = app.writeJSON(w, http.StatusCreated, envelope{"user": user}, nil)
if err != nil {
    app.serverErrorResponse(w, r, err)
}
}

```

Before we try this out, we also need to add the new **POST /v1/users** endpoint to our `cmd/api/routes.go` file. Like so:

```

File: cmd/api/routes.go

package main

...

func (app *application) routes() http.Handler {
    router := httprouter.New()

    router.NotFound = http.HandlerFunc(app.notFoundResponse)
    router.MethodNotAllowed = http.HandlerFunc(app.methodNotAllowedResponse)

    router.HandleFunc(http.MethodGet, "/v1/healthcheck", app.healthcheckHandler)

    router.HandleFunc(http.MethodGet, "/v1/movies", app.listMoviesHandler)
    router.HandleFunc(http.MethodPost, "/v1/movies", app.createMovieHandler)
    router.HandleFunc(http.MethodGet, "/v1/movies/:id", app.showMovieHandler)
    router.HandleFunc(http.MethodPatch, "/v1/movies/:id", app.updateMovieHandler)
    router.HandleFunc(http.MethodDelete, "/v1/movies/:id", app.deleteMovieHandler)

    // Add the route for the POST /v1/users endpoint.
    router.HandleFunc(http.MethodPost, "/v1/users", app.registerUserHandler)

    return app.recoverPanic(app.rateLimit(router))
}

```

Once that's done, make sure all the files are saved and fire up the API.

Then go ahead and make a request to the **POST /v1/users** endpoint to register a new user with the email address `alice@example.com`. You should get a **201 Created** response displaying the details for the user, similar to this:


```
$ BODY='{"name": "Alice Smith", "email": "alice@example.com", "password": "pa55word"}'
$ curl -i -d "$BODY" localhost:4000/v1/users
HTTP/1.1 201 Created
Content-Type: application/json
Date: Mon, 15 Mar 2021 14:42:58 GMT
Content-Length: 152

{
  "user": {
    "id": 1,
    "created_at": "2021-03-15T15:42:58+01:00",
    "name": "Alice Smith",
    "email": "alice@example.com",
    "activated": false
  }
}
```

Hint: If you're following along, remember the password you used in the request above — you'll need it later!

That's looking good. We can see from the status code that the user record has been successfully created, and in the JSON response we can see the system-generated information for the new user — including the user's ID and activation status.

If you take a look at your PostgreSQL database, you should also see the new record in the `users` table. Similar to this:

```
$ psql $GREENLIGHT_DB_DSN
Password for user greenlight:
psql (13.1 (Ubuntu 13.1-1.pgdg20.04+1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression: off)
Type "help" for help.

greenlight=> SELECT * FROM users;
 id |      created_at      | name      | email      | password_hash      | activated | version
-----+-----+-----+-----+-----+-----+-----
  1 | 2021-04-11 14:29:45+02 | Alice Smith | alice@example.com | \x24326124313224526157784d67356d... | f        |      1
(1 row)
```

Note: The `psql` tool always displays `bytea` values as a hex-encoded string. So the `password_hash` field in the output above displays a *hex-encoding* of the bcrypt hash. If you want, you can run the following query to append the regular string version to the table too: `SELECT *, encode(password_hash, 'escape') FROM users`.

OK, let's try making another request to our API but with some invalid user details. This time our validation checks will kick in and the client should receive the relevant error messages. For example:

```
$ BODY='{"name": "", "email": "bob@invalid.", "password": "pass"}'
$ curl -d "$BODY" localhost:4000/v1/users
{
  "error": {
    "email": "must be a valid email address",
    "name": "must be provided",
    "password": "must be at least 8 bytes long"
  }
}
```

Lastly, try registering a second account for `alice@example.com`. This time you should get a validation error containing an *“a user with this email address already exists”* message, like so:

```
$ BODY='{"name": "Alice Jones", "email": "alice@example.com", "password": "pa55word"}'
$ curl -i -d "$BODY" localhost:4000/v1/users
HTTP/1.1 422 Unprocessable Entity
Cache-Control: no-store
Content-Type: application/json
Date: Wed, 30 Dec 2020 14:22:06 GMT
Content-Length: 78

{
  "error": {
    "email": "a user with this email address already exists"
  }
}
```

If you want, you can also try sending some requests using alternative casings of `alice@example.com` — such as `ALICE@example.com` or `Alice@Example.com`. Because the `email` column in our database has the type `citext`, these alternative versions will be successfully identified as duplicates too.

Additional Information

Email case-sensitivity

Let’s talk quickly about email addresses case-sensitivity in a bit more detail.

- Thanks to the specifications in [RFC 2821](#), the domain part of an email address (`username@domain`) is case-insensitive. This means we can be confident that the real-life user behind `alice@example.com` is the same person as `alice@EXAMPLE.COM`.
- The username part of an email address *may or may not* be case-sensitive — it depends on the email provider. Almost every major email provider treats the username as case-

insensitive, but it is not absolutely guaranteed. All we can say here is that the real-life user behind the address `alice@example.com` is *very probably* (but not definitely) the same as `ALICE@example.com`.

So, what does this mean for our application?

From a security point of view, we should always store the email address using the exact casing provided by the user during registration, and **we should send them emails using that exact casing only**. If we don't, there is a risk that emails could be delivered to the wrong real-life user. It's particularly important to be aware of this in any workflows that use email for authentication purposes, such as a password-reset workflow.

However, because `alice@example.com` and `ALICE@example.com` are *very probably* the same user, we should generally treat email addresses as case-insensitive for comparison purposes.

In our registration workflow, using a case-insensitive comparison prevents users from accidentally (or intentionally) registering multiple accounts by just using different casing. And from a user-experience point-of-view, in workflows like login, activation or password resets, it's more forgiving for users if we don't require them to submit their request with exactly the same email casing that they used when registering.

User enumeration

It's important to be aware that our registration endpoint is vulnerable to *user enumeration*. For example, if an attacker wants to know whether `alice@example.com` has an account with us, all they need to do is send a request like this:

```
$ BODY='{"name": "Alice Jones", "email": "alice@example.com", "password": "pa55word"}'
$ curl -d "$BODY" localhost:4000/v1/users
{
  "error": {
    "email": "a user with this email address already exists"
  }
}
```

And they have the answer right there. We're explicitly telling the attacker that `alice@example.com` is already a user.

So, what are the risks of leaking this information?

The first, most obvious, risk relates to user privacy. For services that are sensitive or confidential you probably don't want to make it obvious who has an account. The second risk is that it makes it easier for an attacker to compromise a user's account. Once they

know a user's email address, they can potentially:

- Target the user with social engineering or another type of tailored attack.
- Search for the email address in leaked password tables, and try those same passwords on our service.

Preventing enumeration attacks typically requires two things:

1. Making sure that the response sent to the client is always exactly the same, irrespective of whether a user exists or not. Generally, this means changing your response wording to be ambiguous, and notifying the user of any problems in a side-channel (such as sending them an email to inform them that they already have an account).
2. Making sure that the *time taken* to send the response is always the same, irrespective of whether a user exists or not. In Go, this generally means offloading work to a background goroutine.

Unfortunately, these mitigations tend to increase the complexity of your application and add friction and obscurity to your workflows. For all your regular users who are not attackers, they're a negative from a UX point of view. You have to ask: *is it worth the trade-off?*

There are a few things to think about when answering this question. How important is user privacy in your application? How attractive (high-value) is a compromised account to an attacker? How important is it to reduce friction in your user workflows? The answers to those questions will vary from project-to-project, and will help form the basis for your decision.

It's worth noting that many big-name services, including Twitter, GitHub and Amazon, don't prevent user enumeration (at least not on their registration pages). I'm not suggesting that this makes it OK — just that those companies have decided that the additional friction for the user is worse than the privacy and security risks *in their specific case*.

Sending Emails

In this section of the book we're going to inject some interactivity into our API, and adapt our `registerUserHandler` so that it sends the user a welcome email after they successfully register.

In the process of doing this we're going to cover a few interesting topics. You'll learn:

- How to use the `Mailtrap` SMTP service to send and monitor test emails during development.
- How to use the `html/template` package and Go's *embedded files* functionality to create dynamic and easy-to-manage templates for your email content.
- How to create a reusable `internal/mailer` package for sending emails from your application.
- How to implement a pattern for sending emails in background goroutines, and how to wait for these to complete during a graceful shutdown.

SMTP Server Setup

In order to develop our email sending functionality, we'll need access to a SMTP (Simple Mail Transfer Protocol) server that we can safely use for testing purposes.

There are a huge number of SMTP service providers (such as Postmark, Sendgrid or Amazon SES) that we *could* use to send our emails — or you can even install and run your own SMTP server. But in this book we're going to use [Mailtrap](#).

The reason for using Mailtrap is because it's a specialist service for *sending emails during development and testing*. Essentially, it delivers all emails to an inbox that you can access, instead of sending them to the actual recipient.

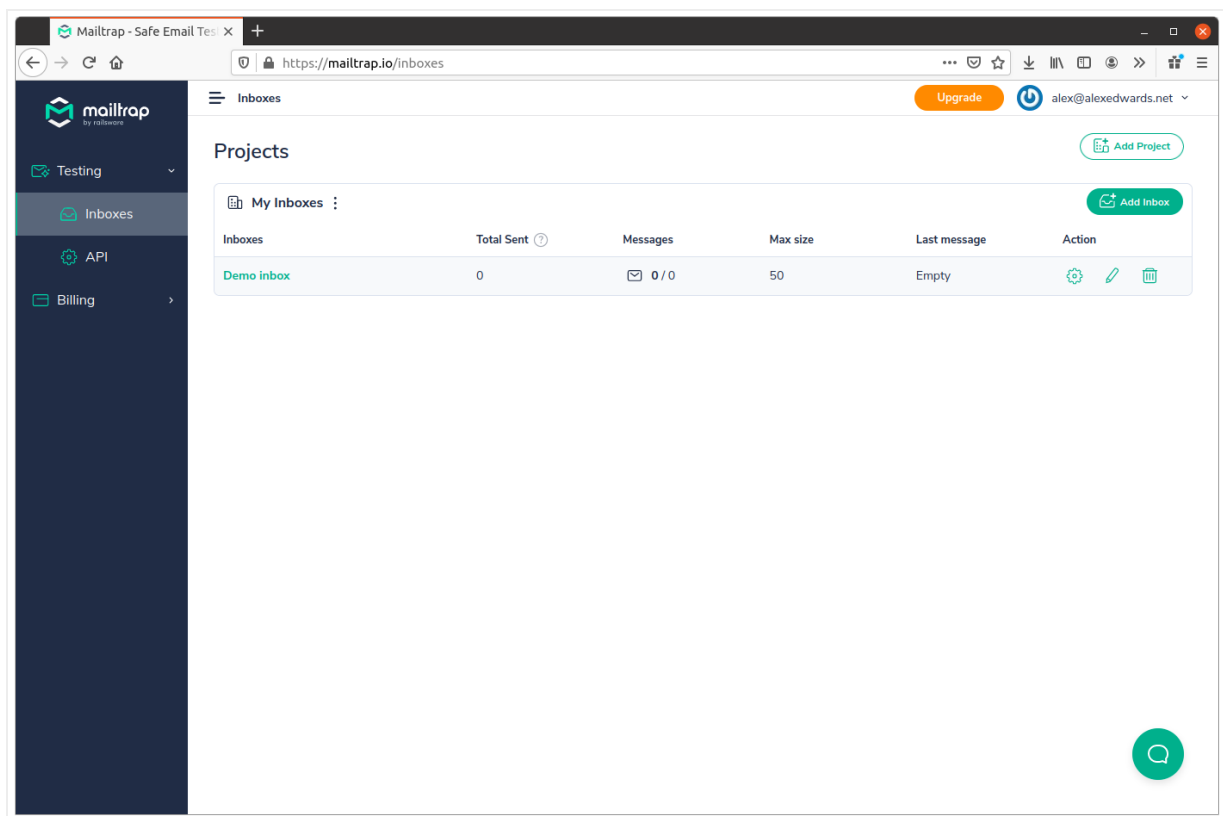
I've got no affiliation with the company — I just find that the service works well and is simple to use. They also offer a 'free forever' plan, which should be sufficient for anyone who is coding-along with this book.

Note: If you already have your own SMTP server, or there is an alternative SMTP provider that you want to use instead, that's absolutely fine. Feel free to skip ahead to the next chapter.

Setting up Mailtrap

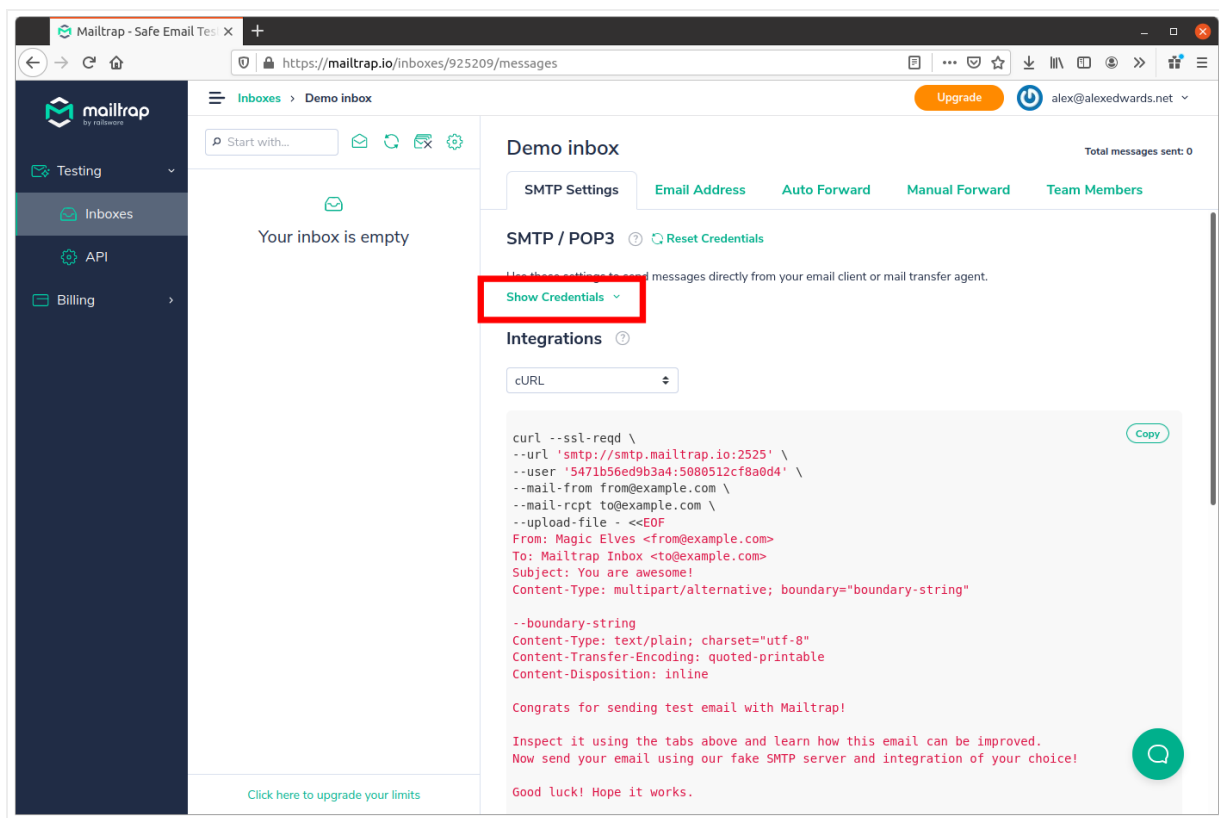
To setup a Mailtrap account, head to the [signup page](#) where you can register using either your email address or your Google or GitHub accounts (if you have them).

Once you're registered and logged in, you should see a page listing your available *inboxes*, similar to the screenshot below.

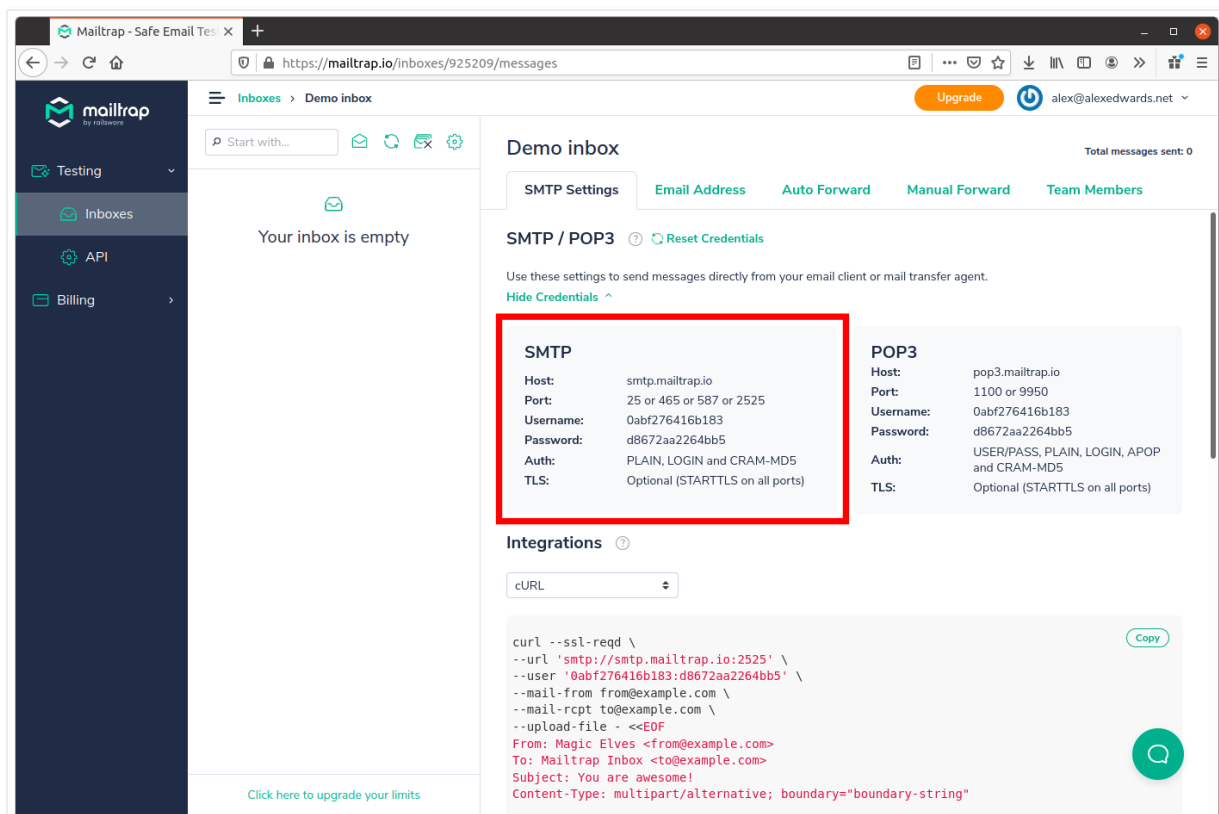


Every Mailtrap account comes with one free inbox, which by default is called either **Demo inbox** or **My inbox** (depending on when your Mailtrap account was created). You can change the name if you want by clicking the pencil icon under **Actions**.

If you go ahead and click through to that inbox, you should see that it's currently empty and contains no emails, similar to this:



Each inbox has its own set of SMTP credentials, which you can display by clicking the **Show Credentials** link (highlighted by the red box in the screenshot above). This will expand to display the inbox SMTP credentials, similar to the screenshot below.



Basically, any emails that you send using these SMTP credentials will end up in this inbox, instead of being sent to the actual recipient.

If you're following along, make a note of the credentials displayed on your screen (or just keep the browser tab open) — you'll need them in the next chapter.

Note: The credentials in the screenshot above have been reset and are no longer valid, so please don't try to use them!

Creating Email Templates

To start with, we'll keep the content of the welcome email really simple, with a short message to let the user know that their registration was successful and confirmation of their ID number. Similar to this:

```
Hi,  
  
Thanks for signing up for a Greenlight account. We're excited to have you on board!  
  
For future reference, your user ID number is 123.  
  
Thanks,  
  
The Greenlight Team
```

Note: Including the user ID probably isn't something you'd normally do in a welcome email, but it's a simple way for us to demonstrate how to include *dynamic data* in emails — rather than just static content.

There are several different approaches we could take to define and manage the content for this email, but a convenient and flexible way is to use Go's templating functionality from the [html/template](#) package.

If you're following along, begin by creating a new `internal/mailer/templates` folder in your project directory and then add a `user_welcome.tmpl` file. Like so:

```
$ mkdir -p internal/mailer/templates  
$ touch internal/mailer/templates/user_welcome.tmpl
```

Inside this file we're going to define three *named templates* to use as part of our welcome email:

- A `"subject"` template containing the subject line for the email.
- A `"plainBody"` template containing the plain-text variant of the email message body.
- A `"htmlBody"` template containing the HTML variant of the email message body.

Go ahead and update the `internal/mailer/templates/user_welcome.tmpl` file to include the content for these templates:

File: internal/mailer/templates/user_welcome.tmpl

```
{{define "subject"}}Welcome to Greenlight!{{end}}

{{define "plainBody"}}
Hi,

Thanks for signing up for a Greenlight account. We're excited to have you on board!

For future reference, your user ID number is {{.ID}}.

Thanks,

The Greenlight Team
{{end}}

{{define "htmlBody"}}
<!doctype html>
<html>

<head>
  <meta name="viewport" content="width=device-width" />
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>

<body>
  <p>Hi,</p>
  <p>Thanks for signing up for a Greenlight account. We're excited to have you on board!</p>
  <p>For future reference, your user ID number is {{.ID}}.</p>
  <p>Thanks,</p>
  <p>The Greenlight Team</p>
</body>

</html>
{{end}}
```

If you've read *Let's Go*, this template syntax and structure should look very familiar to you and we won't dwell on the details again here. But basically:

- We've defined the three *named templates* using the `{{define "..."}}...{{end}}` tags.
- You can render dynamic data in these templates via the `.` character (referred to as *dot*). In the next chapter we'll pass a `User` struct to the templates as dynamic data, which means that we can then render the user's ID using the tag `{{.ID}}` in the templates.

Note: If you need to frequently change the text of emails or require them to be user-editable, then it might be appropriate to store these templates as strings in your database instead. But I've found that storing them in a file, like we are here, is a less complicated approach and a good starting point for most projects.

Sending a Welcome Email

Now that we have the content for the welcome email written, let's create the code to send it.

To send emails we *could* use Go's [net/smtp](#) package from the standard library. But unfortunately it's been frozen for a few years, and doesn't support some of the features that you might need in more advanced use-cases, such as the ability to add attachments.

So instead, I recommend using the third-party [go-mail/mail](#) package to help send email. It's well tested, has good documentation, and a very clear and useable API.

If you're coding-along, please use `go get` to download version `v2.3.0` of this package:

```
$ go get github.com/go-mail/mail/v2@v2.3.0
go: downloading github.com/go-mail/mail/v2 v2.3.0
go: downloading gopkg.in/alexcesaro/quotedprintable.v3 v3.0.0-20150716171945-2caba252f4dc
go get: added github.com/go-mail/mail/v2 v2.3.0
go get: added gopkg.in/alexcesaro/quotedprintable.v3 v3.0.0-20150716171945-2caba252f4dc
```

Creating an email helper

Rather than writing all the code for sending the welcome email in our `registerUserHandler`, in this chapter we're going to create a new `internal/mailer` package which wraps up the logic for parsing our email templates and sending emails.

In addition to that, we're also going to use Go's embedded files functionality, so that the email template files will be *built into our binary* when we create it later. This is really nice because it means we won't have to deploy these template files separately to our production server.

It's probably easiest to demonstrate how this works by getting straight into the code and talking through the details as we go.

Let's begin by creating a new `internal/mailer/mailer.go` file:

```
$ touch internal/mailer/mailer.go
```

And then go ahead and add the following code:

File: internal/mailer/mailer.go

```
package mailer

import (
    "bytes"
    "embed"
    "html/template"
    "time"

    "github.com/go-mail/mail/v2"
)

// Below we declare a new variable with the type embed.FS (embedded file system) to hold
// our email templates. This has a comment directive in the format `//go:embed <path>`
// IMMEDIATELY ABOVE it, which indicates to Go that we want to store the contents of the
// ./templates directory in the templateFS embedded file system variable.
// ↓↓↓↓

//go:embed "templates"
var templateFS embed.FS

// Define a Mailer struct which contains a mail.Dialer instance (used to connect to a
// SMTP server) and the sender information for your emails (the name and address you
// want the email to be from, such as "Alice Smith <alice@example.com>").
type Mailer struct {
    dialer *mail.Dialer
    sender string
}

func New(host string, port int, username, password, sender string) Mailer {
    // Initialize a new mail.Dialer instance with the given SMTP server settings. We
    // also configure this to use a 5-second timeout whenever we send an email.
    dialer := mail.NewDialer(host, port, username, password)
    dialer.Timeout = 5 * time.Second

    // Return a Mailer instance containing the dialer and sender information.
    return Mailer{
        dialer: dialer,
        sender: sender,
    }
}

// Define a Send() method on the Mailer type. This takes the recipient email address
// as the first parameter, the name of the file containing the templates, and any
// dynamic data for the templates as an interface{} parameter.
func (m Mailer) Send(recipient, templateFile string, data interface{}) error {
    // Use the ParseFS() method to parse the required template file from the embedded
    // file system.
    tmpl, err := template.New("email").ParseFS(templateFS, "templates/"+templateFile)
    if err != nil {
        return err
    }

    // Execute the named template "subject", passing in the dynamic data and storing the
    // result in a bytes.Buffer variable.
    subject := new(bytes.Buffer)
    err = tmpl.ExecuteTemplate(subject, "subject", data)
    if err != nil {
        return err
    }

    // Follow the same pattern to execute the "plainBody" template and store the result
    // in the plainBody variable.
    plainBody := new(bytes.Buffer)
```

```

err = tmpl.ExecuteTemplate(plainBody, "plainBody", data)
if err != nil {
    return err
}

// And likewise with the "htmlBody" template.
htmlBody := new(bytes.Buffer)
err = tmpl.ExecuteTemplate(htmlBody, "htmlBody", data)
if err != nil {
    return err
}

// Use the mail.NewMessage() function to initialize a new mail.Message instance.
// Then we use the SetHeader() method to set the email recipient, sender and subject
// headers, the SetBody() method to set the plain-text body, and the AddAlternative()
// method to set the HTML body. It's important to note that AddAlternative() should
// always be called *after* SetBody().
msg := mail.NewMessage()
msg.SetHeader("To", recipient)
msg.SetHeader("From", m.sender)
msg.SetHeader("Subject", subject.String())
msg.SetBody("text/plain", plainBody.String())
msg.AddAlternative("text/html", htmlBody.String())

// Call the DialAndSend() method on the dialer, passing in the message to send. This
// opens a connection to the SMTP server, sends the message, then closes the
// connection. If there is a timeout, it will return a "dial tcp: i/o timeout"
// error.
err = m.dialer.DialAndSend(msg)
if err != nil {
    return err
}

return nil
}

```

Using embedded file systems

Before we continue, let's take a quick moment to discuss embedded file systems in more detail, because there are a couple of things that can be confusing when you first encounter them.

- You can only use the `//go:embed` directive on global variables at package level, not within functions or methods. If you try to use it in a function or method, you'll get the error `"go:embed cannot apply to var inside func"` at compile time.
- When you use the directive `//go:embed "<path>"` to create an embedded file system, the path should be *relative to the source code file containing the directive*. So in our case, `//go:embed "templates"` embeds the contents of the directory at `internal/mailler/templates`.
- The embedded file system is rooted in the directory which contains the `//go:embed` directive. So, in our case, to get the `user_welcome.tmpl` file we need to retrieve it from `templates/user_welcome.tmpl` in the embedded file system.

- Paths cannot contain `.` or `..` elements, nor may they begin or end with a `/`. This essentially restricts you to only embedding files that are contained in the same directory (or a subdirectory) as the source code which has the `//go:embed` directive.
- If the path is for a directory, then all files in the directory are recursively embedded, except for files with names that begin with `.` or `_`. If you want to include these files you should use the `*` wildcard character in the path, like `//go:embed "templates/*"`
- You can specify multiple directories and files in one directive. For example:
`//go:embed "images" "styles/css" "favicon.ico"`.
- The path separator should always be a forward slash, even on Windows machines.

Using our mail helper

Now that our email helper package is in place, we need to hook it up to the rest of our code in the `cmd/api/main.go` file. Specifically, we need to do two things:

- Adapt our code to accept the configuration settings for the SMTP server as command-line flags.
- Initialize a new `Mailer` instance and make it available to our handlers via the `application` struct.

If you're following along, please make sure to use your own Mailtrap SMTP server settings from the previous chapter as the default values for the command line flags here — not the exact values I'm using in the code below.

File: `cmd/api/main.go`

```
package main

import (
    "context"
    "database/sql"
    "flag"
    "os"
    "time"

    "greenlight.alexedwards.net/internal/data"
    "greenlight.alexedwards.net/internal/jsonlog"
    "greenlight.alexedwards.net/internal/mailer" // New import

    _ "github.com/lib/pq"
)

const version = "1.0.0"

// Update the config struct to hold the SMTP server settings
```

```

// Update the config struct to hold the SMTP server settings.
type config struct {
    port int
    env string
    db struct {
        dsn string
        maxOpenConns int
        maxIdleConns int
        maxIdleTime string
    }
    limiter struct {
        enabled bool
        rps float64
        burst int
    }
    smtp struct {
        host string
        port int
        username string
        password string
        sender string
    }
}

// Update the application struct to hold a new Mailer instance.
type application struct {
    config config
    logger *jsonlog.Logger
    models data.Models
    mailer mailer.Mailer
}

func main() {
    var cfg config

    flag.IntVar(&cfg.port, "port", 4000, "API server port")
    flag.StringVar(&cfg.env, "env", "development", "Environment (development|staging|production)")

    flag.StringVar(&cfg.db.dsn, "db-dsn", os.Getenv("GREENLIGHT_DB_DSN"), "PostgreSQL DSN")

    flag.IntVar(&cfg.db.maxOpenConns, "db-max-open-conns", 25, "PostgreSQL max open connections")
    flag.IntVar(&cfg.db.maxIdleConns, "db-max-idle-conns", 25, "PostgreSQL max idle connections")
    flag.StringVar(&cfg.db.maxIdleTime, "db-max-idle-time", "15m", "PostgreSQL max connection idle time")

    flag.BoolVar(&cfg.limiter.enabled, "limiter-enabled", true, "Enable rate limiter")
    flag.Float64Var(&cfg.limiter.rps, "limiter-rps", 2, "Rate limiter maximum requests per second")
    flag.IntVar(&cfg.limiter.burst, "limiter-burst", 4, "Rate limiter maximum burst")

    // Read the SMTP server configuration settings into the config struct, using the
    // Mailtrap settings as the default values. IMPORTANT: If you're following along,
    // make sure to replace the default values for smtp-username and smtp-password
    // with your own Mailtrap credentials.
    flag.StringVar(&cfg.smtp.host, "smtp-host", "smtp.mailtrap.io", "SMTP host")
    flag.IntVar(&cfg.smtp.port, "smtp-port", 25, "SMTP port")
    flag.StringVar(&cfg.smtp.username, "smtp-username", "0abf276416b183", "SMTP username")
    flag.StringVar(&cfg.smtp.password, "smtp-password", "d8672aa2264bb5", "SMTP password")
    flag.StringVar(&cfg.smtp.sender, "smtp-sender", "Greenlight <no-reply@greenlight.alexedwards.net>", "SMTP sender")

    flag.Parse()

    logger := jsonlog.New(os.Stdout, jsonlog.LevelInfo)

    db, err := openDB(cfg)
    if err != nil {
        logger.PrintFatal(err, nil)
    }
    defer db.Close()

```



```

logger.Println("database connection pool established", nil)

// Initialize a new Mailer instance using the settings from the command line
// flags, and add it to the application struct.
app := &Application{
    config: cfg,
    logger: logger,
    models: data.NewModels(db),
    mailer: mailer.New(cfg.smtp.host, cfg.smtp.port, cfg.smtp.username, cfg.smtp.password, cfg.smtp.sender),
}

err = app.serve()
if err != nil {
    logger.Fatalf(err, nil)
}
}

...

```

And then the final thing we need to do is update our `registerUserHandler` to actually send the email, which we can do like so:

```

File: cmd/api/users.go

package main

...

func (app *application) registerUserHandler(w http.ResponseWriter, r *http.Request) {

    ... // Nothing above here needs to change.

    // Call the Send() method on our Mailer, passing in the user's email address,
    // name of the template file, and the User struct containing the new user's data.
    err = app.mailer.Send(user.Email, "user_welcome.tmpl", user)
    if err != nil {
        app.serverErrorResponse(w, r, err)
        return
    }

    err = app.writeJSON(w, http.StatusCreated, envelope{"user": user}, nil)
    if err != nil {
        app.serverErrorResponse(w, r, err)
    }
}

```

Alright, let's try this out!

Run the application, then in another terminal use `curl` to register a brand-new user with the email address `bob@example.com`:

```
$ BODY='{"name": "Bob Jones", "email": "bob@example.com", "password": "pa55word"}'
$ curl -w '\nTime: %{time_total}\n' -d "$BODY" localhost:4000/v1/users
{
  "user": {
    "id": 3,
    "created_at": "2021-04-11T20:26:22+02:00",
    "name": "Bob Jones",
    "email": "bob@example.com",
    "activated": false
  }
}

Time: 2.331957
```

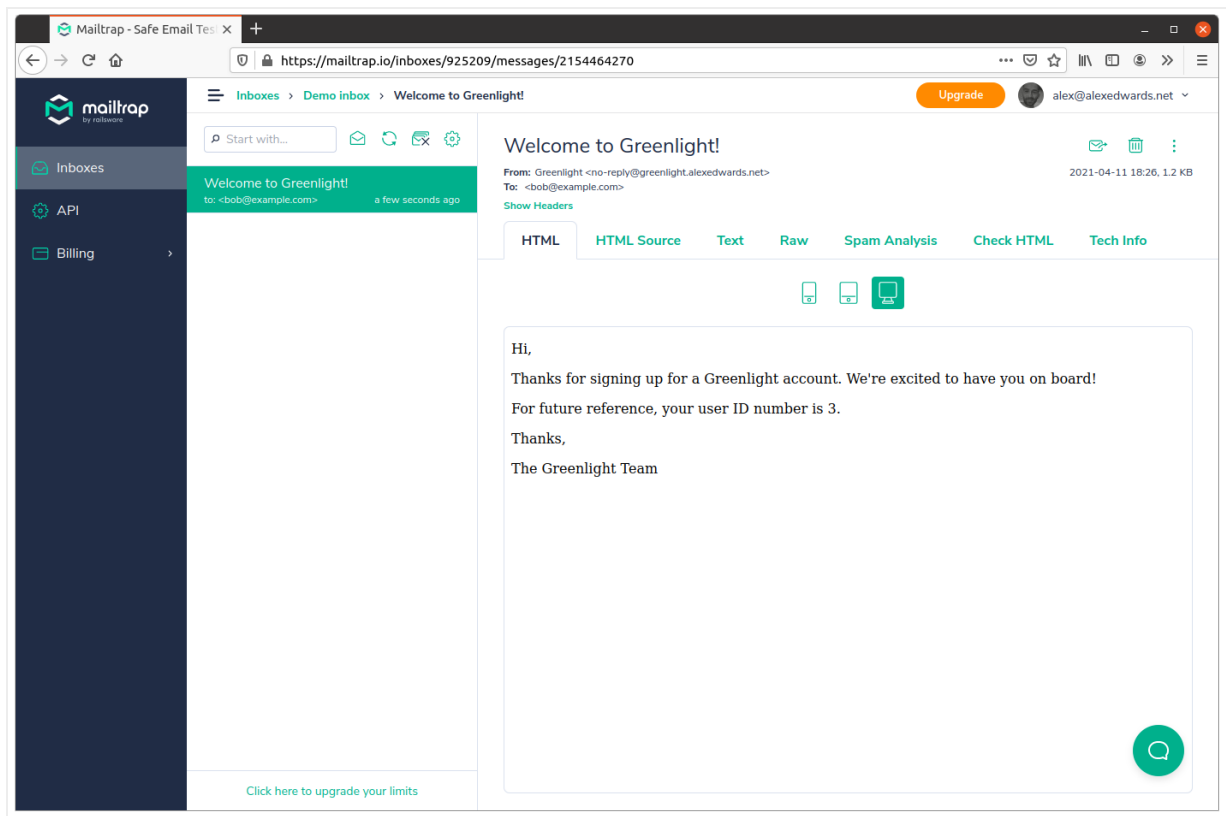
If everything is set up correctly you should get a **201 Created** response containing the new user details, similar to the response above.

Important: If you receive a **"dial tcp: connect: connection refused"** error this means that your application could not connect to the Mailtrap SMTP server. Please double check that your Mailtrap credentials are correct, or try using port **2525** instead of port **25**.

Note: I've used the **-w** flag again to display the total time taken for the request to complete, which in my case was ≈ 2.3 seconds. That's quite a long time for a HTTP request to complete, and we'll speed it up in the next chapter by sending the welcome email in a background goroutine.

Checking the email in Mailtrap

If you've been following along and are using the Mailtrap SMTP server credentials, when you go back to your account you should see now the welcome email for **bob@example.com** in your **Demo inbox**, like so:



If you want, you can also click on the **Text** tab to see the plain-text version of the email, and the **Raw** tab to see the complete email including headers.

Wrapping this up, we covered quite a lot of ground in this chapter. But the nice thing about the pattern we've built is that it's easy to extend. If we want to send other emails from our application in the future, we can simply make an additional file in our `internal/mailer/templates` folder with the email content, and then send it from our handlers in the same way that we have done here.

Additional Information

Retrying email send attempts

If you want, you can make the email sending process a bit more robust by adding some basic 'retry' functionality to the `Mailer.Send()` method. For example:

```

func (m Mailer) Send(recipient, templateFile string, data interface{}) error {
    ...

    // Try sending the email up to three times before aborting and returning the final
    // error. We sleep for 500 milliseconds between each attempt.
    for i := 1; i <= 3; i++ {
        err = m.dialer.DialAndSend(msg)
        // If everything worked, return nil.
        if nil == err {
            return nil
        }

        // If it didn't work, sleep for a short time and retry.
        time.Sleep(500 * time.Millisecond)
    }

    return err
}

```

Hint: In the code above we're using the clause `if nil == err` to check if the send was successful, rather than `if err == nil`. They're functionally equivalent, but having `nil` as the first item in the clause makes it a bit visually jarring and less likely to be confused with the far more common `if err != nil` clause.

This retry functionality is a relatively simple addition to our code, but it helps to increase the probability that emails are successfully sent in the event of transient network issues. If you're sending emails in a background process (like we will do in the next chapter), you might want to make the sleep duration even longer here, as it won't materially impact the client and gives more time for transient issues to resolve.

Sending Background Emails

As we mentioned briefly in the last chapter, sending the welcome email from the `registerUserHandler` method adds quite a lot of latency to the total request/response round-trip for the client.

One way we could reduce this latency is by sending the email in a *background goroutine*. This would effectively ‘decouple’ the task of sending an email from the rest of the code in our `registerUserHandler`, and means that we could return a HTTP response to the client without waiting for the email sending to complete.

At its very simplest, we could adapt our handler to execute the email send in a background goroutine like this:

File: cmd/api/users.go

```
package main
...

func (app *application) registerUserHandler(w http.ResponseWriter, r *http.Request) {
    ...

    // Launch a goroutine which runs an anonymous function that sends the welcome email.
    go func() {
        err = app.mailer.Send(user.Email, "user_welcome.tmpl", user)
        if err != nil {
            // Importantly, if there is an error sending the email then we use the
            // app.logger.PrintError() helper to manage it, instead of the
            // app.serverErrorResponse() helper like before.
            app.logger.PrintError(err, nil)
        }
    }()

    // Note that we also change this to send the client a 202 Accepted status code.
    // This status code indicates that the request has been accepted for processing, but
    // the processing has not been completed.
    err = app.writeJSON(w, http.StatusAccepted, envelope{"user": user}, nil)
    if err != nil {
        app.serverErrorResponse(w, r, err)
    }
}
```

When this code is executed now, a new ‘background’ goroutine will be launched for sending the welcome email. The code in this background goroutine will be executed *concurrently* with the subsequent code in our `registerUserHandler`, which means we are no longer waiting for the email to be sent before we return a JSON response to the client. Most likely,

the background goroutine will still be executing its code long after the `registerUserHandler` has returned.

There are a couple of things I'd like to emphasize here:

- We use the `app.logger.PrintError()` helper to manage any errors in our background goroutine. This is because by the time we encounter the errors, the client will probably have already been sent a `202 Accepted` response by our `writeJSON()` helper.

Note that we don't want to use the `app.serverErrorResponse()` helper to handle any errors in our background goroutine, as that would result in us trying to write a *second HTTP response* and getting a `"http: superfluous response.WriteHeader call"` error from our `http.Server` at runtime.

- The code running in the background goroutine forms a *closure* over the `user` and `app` variables. It's important to be aware that these 'closed over' variables are *not scoped to the background goroutine*, which means that any changes you make to them will be reflected in the rest of your codebase. For a simple example of this, see the following [playground code](#).

In our case we aren't changing the value of these variables in any way, so this behavior won't cause us any issues. But it is important to keep in mind.

OK, let's try this out!

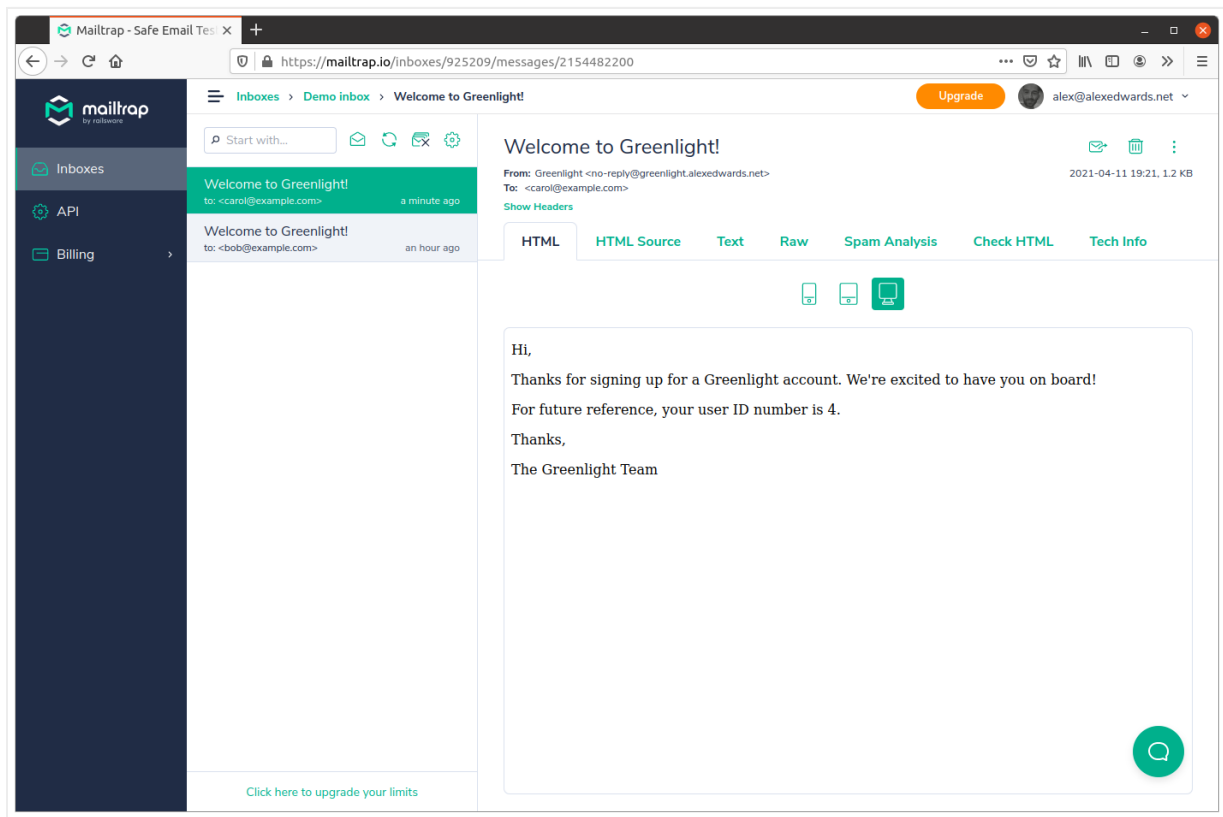
Restart the API, then go ahead and register another new user with the email address `carol@example.com`. Like so:

```
$ BODY='{ "name": "Carol Smith", "email": "carol@example.com", "password": "pa55word" }'
$ curl -w '\nTime: %{time_total}\n' -d "$BODY" localhost:4000/v1/users
{
  "user": {
    "id": 4,
    "created_at": "2021-04-11T21:21:12+02:00",
    "name": "Carol Smith",
    "email": "carol@example.com",
    "activated": false
  }
}

Time: 0.268639
```

This time, you should see that the time taken to return the response is much faster — in my case 0.27 seconds compared to the previous 2.33 seconds.

And if you take a look at your Mailtrap inbox, you should see that the email for `carol@example.com` has been delivered correctly. Like so:



Recovering panics

It's important to bear in mind that any panic which happens in this background goroutine will not be automatically recovered by our `recoverPanic()` middleware or Go's `http.Server`, and will cause our whole application to terminate.

In very simple background goroutines (like the ones we've been using so far), this is less of a worry. But the code involved in sending an email is quite complex (including calls to a third-party package) and the risk of a runtime panic is non-negligible. So we need to make sure that any panic in this background goroutine is manually recovered, using a similar pattern to the one in our `recoverPanic()` middleware.

I'll demonstrate.

Reopen your `cmd/api/users.go` file and update the `registerUserHandler` like so:

File: cmd/api/users.go

```
package main

import (
    "errors"
    "fmt" // New import
    "net/http"

    "greenlight.alexedwards.net/internal/data"
    "greenlight.alexedwards.net/internal/validator"
)

func (app *application) registerUserHandler(w http.ResponseWriter, r *http.Request) {
    ...

    // Launch a background goroutine to send the welcome email.
    go func() {
        // Run a deferred function which uses recover() to catch any panic, and log an
        // error message instead of terminating the application.
        defer func() {
            if err := recover(); err != nil {
                app.logger.PrintError(fmt.Errorf("%s", err), nil)
            }
        }()

        // Send the welcome email.
        err = app.mailer.Send(user.Email, "user_welcome.tmpl", user)
        if err != nil {
            app.logger.PrintError(err, nil)
        }
    }()

    err = app.writeJSON(w, http.StatusAccepted, envelope{"user": user}, nil)
    if err != nil {
        app.serverErrorResponse(w, r, err)
    }
}
```

Using a helper function

If you need to execute a lot of background tasks in your application, it can get tedious to keep repeating the same panic recovery code — and there's a risk that you might forget to include it altogether.

To help take care of this, it's possible to create a simple helper function which wraps the panic recovery logic. If you're following along, open your `cmd/api/helpers.go` file and create a new `background()` helper method as follows:

File: cmd/api/helpers.go

```
package main

...

// The background() helper accepts an arbitrary function as a parameter.
func (app *application) background(fn func()) {
    // Launch a background goroutine.
    go func() {
        // Recover any panic.
        defer func() {
            if err := recover(); err != nil {
                app.logger.PrintError(fmt.Errorf("%s", err), nil)
            }
        }()

        // Execute the arbitrary function that we passed as the parameter.
        fn()
    }()
}
```

This `background()` helper leverages the fact that Go has [first-class functions](#), which means that functions can be *assigned to variables* and *passed as parameters* to other functions.

In this case, we've set up the `background()` helper so that it accepts any function with the signature `func()` as a parameter and stores it in the variable `fn`. It then spins up a background goroutine, uses a deferred function to recover any panics and log the error, and then executes the function itself by calling `fn()`.

Now that this is in place, let's update our `registerUserHandler` to use it like so:

File: cmd/api/users.go

```
package main

import (
    "errors"
    "net/http"

    "greenlight.alexedwards.net/internal/data"
    "greenlight.alexedwards.net/internal/validator"
)

func (app *application) registerUserHandler(w http.ResponseWriter, r *http.Request) {
    ...

    // Use the background helper to execute an anonymous function that sends the welcome
    // email.
    app.background(func() {
        err = app.mailer.Send(user.Email, "user_welcome.tmpl", user)
        if err != nil {
            app.logger.PrintError(err, nil)
        }
    })

    err = app.writeJSON(w, http.StatusOK, envelope{"user": user}, nil)
    if err != nil {
        app.serverErrorResponse(w, r, err)
    }
}
```

Let's double-check that this is still working. Restart the API, then create another new user with the email address **dave@example.com**:

```
$ BODY='{ "name": "Dave Smith", "email": "dave@example.com", "password": "pa55word" }'
$ curl -w '\nTime: %{time_total}\n' -d "$BODY" localhost:4000/v1/users
{
  "user": {
    "id": 5,
    "created_at": "2021-04-11T21:33:07+02:00",
    "name": "Dave Smith",
    "email": "dave@example.com",
    "activated": false
  }
}

Time: 0.267692
```

If everything is set up correctly, you'll now see the corresponding email appear in your Mailtrap inbox again.

Mailtrap - Safe Email Test

https://mailtrap.io/inboxes/925209/messages/2154486218

mailtrap

by hollaxone

Inboxes

API

Billing

Start with...

Welcome to Greenlight!

to: <dave@example.com>

a few seconds ago

Welcome to Greenlight!

to: <carol@example.com>

12 minutes ago

Welcome to Greenlight!

to: <bob@example.com>

an hour ago

Click here to upgrade your limits

Upgrade

alex@alexedwards.net

Welcome to Greenlight!

From: Greenlight <no-reply@greenlight.alexedwards.net>

To: <dave@example.com>

2021-04-11 19:33, 1.2 KB

Show Headers

HTML

HTML Source

Text

Raw

Spam Analysis

Check HTML

Tech Info

Hi,

Thanks for signing up for a Greenlight account. We're excited to have you on board!

For future reference, your user ID number is 5.

Thanks,

The Greenlight Team

Graceful Shutdown of Background Tasks

Sending our welcome email in the background is working well, but there's still an issue we need to address.

When we initiate a graceful shutdown of our application, it *won't wait for any background goroutines that we've launched to complete*. So — if we happen to shutdown our server at an unlucky moment — it's possible that a new client will be created on our system but they will never be sent their welcome email.

Fortunately, we can prevent this by using Go's `sync.WaitGroup` functionality to coordinate the graceful shutdown and our background goroutines.

An introduction to `sync.WaitGroup`

When you want to wait for a collection of goroutines to finish their work, the principal tool to help with this is the `sync.WaitGroup` type.

The way that it works is conceptually a bit like a 'counter'. Each time you launch a background goroutine you can increment the counter by 1, and when each goroutine finishes, you then decrement the counter by 1. You can then monitor the counter, and when it equals zero you know that all your background goroutines have finished.

Let's take a quick look at a standalone example of how `sync.WaitGroup` works in practice.

In the code below, we'll launch five goroutines that print out `"hello from a goroutine"`, and use `sync.WaitGroup` to wait for them all to complete before the program exits.

```

package main

import (
    "fmt"
    "sync"
)

func main() {
    // Declare a new WaitGroup.
    var wg sync.WaitGroup

    // Execute a loop 5 times.
    for i := 1; i <= 5; i++ {
        // Increment the WaitGroup counter by 1, BEFORE we launch the background routine.
        wg.Add(1)

        // Launch the background goroutine.
        go func() {
            // Defer a call to wg.Done() to indicate that the background goroutine has
            // completed when this function returns. Behind the scenes this decrements
            // the WaitGroup counter by 1 and is the same as writing wg.Add(-1).
            defer wg.Done()

            fmt.Println("hello from a goroutine")
        }()
    }

    // Wait() blocks until the WaitGroup counter is zero --- essentially blocking until all
    // goroutines have completed.
    wg.Wait()

    fmt.Println("all goroutines finished")
}

```

If you [run the above code](#), you'll see that the output looks like this:

```

hello from a goroutine
hello from a goroutine
hello from a goroutine
hello from a goroutine
hello from a goroutine
all goroutines finished

```

One thing that's important to emphasize here is that we increment the counter with `wg.Add(1)` immediately *before* we launch the background goroutine. If we called `wg.Add(1)` in the background goroutine itself, there is a race condition because `wg.Wait()` could potentially be called *before the counter is even incremented*.

Fixing our application

Let's update our application to incorporate a `sync.WaitGroup` that coordinates our graceful shutdown and background goroutines.

We'll begin in our `cmd/api/main.go` file, and edit the `application` struct to contain a new `sync.WaitGroup`. Like so:

File: `cmd/api/main.go`

```
package main

import (
    "context"
    "database/sql"
    "flag"
    "os"
    "sync" // New import
    "time"

    "greenlight.alexedwards.net/internal/data"
    "greenlight.alexedwards.net/internal/jsonlog"
    "greenlight.alexedwards.net/internal/mailer"

    _ "github.com/lib/pq"
)

...

// Include a sync.WaitGroup in the application struct. The zero-value for a
// sync.WaitGroup type is a valid, useable, sync.WaitGroup with a 'counter' value of 0,
// so we don't need to do anything else to initialize it before we can use it.
type application struct {
    config config
    logger *jsonlog.Logger
    models data.Models
    mailer mailer.Mailer
    wg     sync.WaitGroup
}

...
```

Next let's head to the `cmd/api/helpers.go` file and update the `app.background()` helper so that the `sync.WaitGroup` counter is incremented each time before we launch a background goroutine, and then decremented when it completes.

Like this:

File: cmd/api/helpers.go

```
package main

...

func (app *application) background(fn func()) {
    // Increment the WaitGroup counter.
    app.wg.Add(1)

    // Launch the background goroutine.
    go func() {
        // Use defer to decrement the WaitGroup counter before the goroutine returns.
        defer app.wg.Done()

        defer func() {
            if err := recover(); err != nil {
                app.logger.PrintError(fmt.Errorf("%s", err), nil)
            }
        }()

        fn()
    }()
}
```

Then the final thing we need to do is update our graceful shutdown functionality so that it uses our new `sync.WaitGroup` to wait for any background goroutines before terminating the application. We can do that by adapting our `app.serve()` method like so:

File: cmd/api/server.go

```
package main

...

func (app *application) serve() error {
    srv := &http.Server{
        Addr:      fmt.Sprintf(":%d", app.config.port),
        Handler:   app.routes(),
        IdleTimeout: time.Minute,
        ReadTimeout: 10 * time.Second,
        WriteTimeout: 30 * time.Second,
    }

    shutdownError := make(chan error)

    go func() {
        quit := make(chan os.Signal, 1)
        signal.Notify(quit, syscall.SIGINT, syscall.SIGTERM)
        s := <-quit

        app.logger.PrintInfo("caught signal", map[string]string{
            "signal": s.String(),
        })

        ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
        defer cancel()

        // Call Shutdown() on the server like before, but now we only send on the
        // shutdownError channel if it returns an error.
    }()
```

```

    err := srv.Shutdown(ctx)
    if err != nil {
        shutdownError <- err
    }

    // Log a message to say that we're waiting for any background goroutines to
    // complete their tasks.
    app.logger.Printf("completing background tasks", map[string]string{
        "addr": srv.Addr,
    })

    // Call Wait() to block until our WaitGroup counter is zero --- essentially
    // blocking until the background goroutines have finished. Then we return nil on
    // the shutdownError channel, to indicate that the shutdown completed without
    // any issues.
    app.wg.Wait()
    shutdownError <- nil
}()

app.logger.Printf("starting server", map[string]string{
    "addr": srv.Addr,
    "env": app.config.env,
})

err := srv.ListenAndServe()
if !errors.Is(err, http.ErrServerClosed) {
    return err
}

err = <-shutdownError
if err != nil {
    return err
}

app.logger.Printf("stopped server", map[string]string{
    "addr": srv.Addr,
})

return nil
}

```

To try this out, go ahead and restart the API and then send a request to the **POST /v1/users** endpoint immediately followed by a **SIGTERM** signal. For example:

```

$ BODY='{ "name": "Edith Smith", "email": "edith@example.com", "password": "pa55word"}'
$ curl -d "$BODY" localhost:4000/v1/users & kill -SIGTERM api &

```

When you do this, your server logs should look similar to the output below:

```

$ go run ./cmd/api
{"level":"INFO","time":"2021-04-15T15:06:50Z","message":"database connection pool established"}
{"level":"INFO","time":"2021-04-15T15:06:50Z","message":"starting server","properties":{"addr":":4000","env":"development"}}
{"level":"INFO","time":"2021-04-15T15:07:06Z","message":"caught signal","properties":{"signal":"terminated"}}
{"level":"INFO","time":"2021-04-15T15:07:06Z","message":"completing background tasks","properties":{"addr":":4000"}}
{"level":"INFO","time":"2021-04-15T15:07:08Z","message":"stopped server","properties":{"addr":":4000"}}

```

Notice how the **"completing background tasks"** message is written, then there is a pause

of a couple of seconds while the background email sending completes, followed finally by the "stopped server" message?

This nicely illustrates how the graceful shutdown process waited for the welcome email to be sent (which took about two seconds in my case) before finally terminating the application.

User Activation

At the moment a user can register for an account with our *Greenlight* API, but we don't know for sure that the email address they provided during registration *actually belongs to them*.

So, in this section of the book, we're going to build up the functionality to confirm that a user used their own real email address by including 'account activation' instructions in their welcome email.

There are [several reasons](#) for having an activation step, but the main benefits are that it adds an additional hoop for bots to jump through, and helps prevent abuse by people who register with a fake email address or one that doesn't belong to them.

To give you an overview upfront, the account activation process will work like this:

1. As part of the registration process for a new user we will create a cryptographically-secure random *activation token* that is impossible to guess.
2. We will then store a hash of this activation token in a new `tokens` table, alongside the new user's ID and an expiry time for the token.
3. We will send the original (unhashed) activation token to the user in their welcome email.
4. The user subsequently submits their token to a new `PUT /v1/users/activated` endpoint.
5. If the hash of the token exists in the `tokens` table and hasn't expired, then we'll update the `activated` status for the relevant user to `true`.
6. Lastly, we'll delete the activation token from our `tokens` table so that it can't be used again.

In this section of the book, you'll learn how to:

- Implement a secure 'account activation' workflow which verifies a new user's email address.
- Generate cryptographically-secure random tokens using Go's `crypto/rand` and `encoding/base32` packages.
- Generate fast hashes of data using the `crypto/sha256` package.
- Implement patterns for working with cross-table relationships in your database,

including setting up foreign keys and retrieving related data via SQL **JOIN** queries.

Setting up the Tokens Database Table

Let's begin by creating a new **tokens** table in our database to store the activation tokens for our users. If you're following along, run the following command to create a new pair of migration files:

```
$ migrate create -seq -ext .sql -dir ./migrations create_tokens_table  
/home/alex/Projects/greenlight/migrations/000005_create_tokens_table.up.sql  
/home/alex/Projects/greenlight/migrations/000005_create_tokens_table.down.sql
```

And then add the following SQL statements to the 'up' and 'down' migration files respectively:

File: migrations/000005_create_tokens_table.up.sql

```
CREATE TABLE IF NOT EXISTS tokens (  
  hash bytea PRIMARY KEY,  
  user_id bigint NOT NULL REFERENCES users ON DELETE CASCADE,  
  expiry timestamp(0) with time zone NOT NULL,  
  scope text NOT NULL  
);
```

File: migrations/000005_create_tokens_table.down.sql

```
DROP TABLE IF EXISTS tokens;
```

Let's quickly step through the columns in this new **tokens** table and explain their purpose.

- The **hash** column will contain a SHA-256 hash of the activation token. It's important to emphasize that we will only store a hash of the activation token in our database — not the activation token itself.

We want to hash the token before storing it for the same reason that we bcrypt a user's password — it provides an extra layer of protection if the database is ever compromised or leaked. Because our activation token is going to be a high-entropy random string (128 bits) — rather than something low entropy like a typical user password — **it is sufficient** to use a fast algorithm like SHA-256 to create the hash, instead of a slow algorithm like bcrypt.

- The **user_id** column will contain the ID of the user associated with the token. We use the

REFERENCES `user` syntax to create a **foreign key constraint** against the primary key of our `users` table, which ensures that any value in the `user_id` column has a corresponding `id` entry in our `users` table.

We also use the **ON DELETE CASCADE** syntax to instruct PostgreSQL to *automatically delete all records for a user in our `tokens` table when the parent record in the `users` table is deleted.*

Note: A common alternative to **ON DELETE CASCADE** is **ON DELETE RESTRICT**, which in our case would *prevent* a parent record in the `users` table from being deleted if the user has any tokens in our `tokens` table. If you use **ON DELETE RESTRICT**, you would need to manually delete any tokens for the user *before* you delete the user record itself.

- The `expiry` column will contain the time that we consider a token to be ‘expired’ and no longer valid. Setting a short expiry time is good from a security point-of-view because it helps reduce the window of possibility for a successful brute-force attack against the token. And it also helps in the scenario where the user is *sent a token but doesn’t use it*, and their email account is compromised at a later time. By setting a short time limit, it reduces the time window that the compromised token could be used.

Of course, the security risks here need to be weighed up against usability, and we want the expiry time to be long enough for a user to be able to activate the account at their leisure. In our case, we’ll set the expiry time for our activation tokens to 3 days from the moment the token was created.

- Lastly, the `scope` column will denote what *purpose* the token can be used for. Later in the book we’ll also need to create and store *authentication tokens*, and most of the code and storage requirements for these is exactly the same as for our activation tokens. So instead of creating separate tables (and the code to interact with them), we’ll store them in one table with a value in the `scope` column to restrict the purpose that the token can be used for.

OK, with those explanations out of the way, you should be able to execute the ‘up’ migration with the following command:

```
$ migrate -path=./migrations -database=$GREENLIGHT_DB_DSN up
5/u create_tokens_table (21.568194ms)
```

Creating Secure Activation Tokens

The integrity of our activation process hinges on one key thing: the ‘unguessability’ of the token that we send to the user’s email address. If the token is easy to guess or can be brute-forced, then it would be possible for an attacker to activate a user’s account even if they don’t have access to the user’s email inbox.

Because of this, we want the token to be generated by a *cryptographically secure random number generator* (CSPRNG) and have enough entropy (or *randomness*) that it is impossible to guess. In our case, we’ll create our activation tokens using Go’s `crypto/rand` package and 128-bits (16 bytes) of entropy.

If you’re following along, go ahead and create a new `internal/data/tokens.go` file. This will act as the home for all our logic related to creating and managing tokens over the next couple of chapters.

```
$ touch internal/data/tokens.go
```

Then in this file let’s define a `Token` struct (to represent the data for an individual token), and a `generateToken()` function that we can use to create a new token.

This is another time where it’s probably easiest to jump straight into the code, and describe what’s happening as we go along.

```
File: internal/data/tokens.go

package data

import (
    "crypto/rand"
    "crypto/sha256"
    "encoding/base32"
    "time"
)

// Define constants for the token scope. For now we just define the scope "activation"
// but we'll add additional scopes later in the book.
const (
    ScopeActivation = "activation"
)

// Define a Token struct to hold the data for an individual token. This includes the
// plaintext and hashed versions of the token, associated user ID, expiry time and
// scope.
type Token struct {
    Plaintext string
```

```

    Hash      []byte
    UserID    int64
    Expiry    time.Time
    Scope     string
}

func generateToken(userID int64, ttl time.Duration, scope string) (*Token, error) {
    // Create a Token instance containing the user ID, expiry, and scope information.
    // Notice that we add the provided ttl (time-to-live) duration parameter to the
    // current time to get the expiry time?
    token := &Token{
        UserID: userID,
        Expiry: time.Now().Add(ttl),
        Scope:  scope,
    }

    // Initialize a zero-valued byte slice with a length of 16 bytes.
    randomBytes := make([]byte, 16)

    // Use the Read() function from the crypto/rand package to fill the byte slice with
    // random bytes from your operating system's CSPRNG. This will return an error if
    // the CSPRNG fails to function correctly.
    _, err := rand.Read(randomBytes)
    if err != nil {
        return nil, err
    }

    // Encode the byte slice to a base-32-encoded string and assign it to the token
    // Plaintext field. This will be the token string that we send to the user in their
    // welcome email. They will look similar to this:
    //
    // Y3QMGX3PJ3WLRL2YRTQGQ6KRHU
    //
    // Note that by default base-32 strings may be padded at the end with the =
    // character. We don't need this padding character for the purpose of our tokens, so
    // we use the WithPadding(base32.NoPadding) method in the line below to omit them.
    token.Plaintext = base32.StdEncoding.WithPadding(base32.NoPadding).EncodeToString(randomBytes)

    // Generate a SHA-256 hash of the plaintext token string. This will be the value
    // that we store in the `hash` field of our database table. Note that the
    // sha256.Sum256() function returns an *array* of length 32, so to make it easier to
    // work with we convert it to a slice using the [:] operator before storing it.
    hash := sha256.Sum256([]byte(token.Plaintext))
    token.Hash = hash[:]

    return token, nil
}

```

It's important to point out that the plaintext token strings we're creating here like **Y3QMGX3PJ3WLRL2YRTQGQ6KRHU** are *not* 16 characters long — but rather they have *an underlying entropy of 16 bytes of randomness*.

The length of the plaintext token string itself depends on *how those 16 random bytes are encoded to create a string*. In our case we encode the random bytes to a base-32 string, which results in a string with 26 characters. In contrast, if we encoded the random bytes using hexadecimal (base-16) the string would be 32 characters long instead.

Creating the TokenModel and Validation Checks

OK, let's move on and set up a `TokenModel` type which encapsulates the database interactions with our PostgreSQL `tokens` table. We'll follow a very similar pattern to the `MovieModel` and `UsersModel` again, and we'll implement the following three methods on it:

- `Insert()` to insert a new token record in the database.
- `New()` will be a shortcut method which creates a new token using the `generateToken()` function and then calls `Insert()` to store the data.
- `DeleteAllForUser()` to delete all tokens with a specific scope for a specific user.

We'll also create a new `ValidateTokenPlaintext()` function, which will check that a plaintext token provided by a client in the future is exactly 26 bytes long.

Open up the `internal/data/tokens.go` file again, and add the following code:

File: internal/data/tokens.go

```
package data

import (
    "context" // New import
    "crypto/rand"
    "crypto/sha256"
    "database/sql" // New import
    "encoding/base32"
    "time"

    "greenlight.alexedwards.net/internal/validator" // New import
)

...

// Check that the plaintext token has been provided and is exactly 26 bytes long.
func ValidateTokenPlaintext(v *validator.Validator, tokenPlaintext string) {
    v.Check(tokenPlaintext != "", "token", "must be provided")
    v.Check(len(tokenPlaintext) == 26, "token", "must be 26 bytes long")
}

// Define the TokenModel type.
type TokenModel struct {
    DB *sql.DB
}

// The New() method is a shortcut which creates a new Token struct and then inserts the
// data in the tokens table.
func (m TokenModel) New(userID int64, ttl time.Duration, scope string) (*Token, error) {
    token, err := generateToken(userID, ttl, scope)
    if err != nil {
        return nil, err
    }

    err = m.Insert(token)
    return token, err
}
```



```
// Insert() adds the data for a specific token to the tokens table.
func (m TokenModel) Insert(token *Token) error {
    query := `
        INSERT INTO tokens (hash, user_id, expiry, scope)
        VALUES ($1, $2, $3, $4)`

    args := []interface{}{token.Hash, token.UserID, token.Expiry, token.Scope}

    ctx, cancel := context.WithTimeout(context.Background(), 3*time.Second)
    defer cancel()

    _, err := m.DB.ExecContext(ctx, query, args...)
    return err
}

// DeleteAllForUser() deletes all tokens for a specific user and scope.
func (m TokenModel) DeleteAllForUser(scope string, userID int64) error {
    query := `
        DELETE FROM tokens
        WHERE scope = $1 AND user_id = $2`

    ctx, cancel := context.WithTimeout(context.Background(), 3*time.Second)
    defer cancel()

    _, err := m.DB.ExecContext(ctx, query, scope, userID)
    return err
}
```

And finally, we need to update the `internal/data/models.go` file so that the new `TokenModel` is included in our parent `Models` struct. Like so:

```
File: internal/data/models.go

package data

...

type Models struct {
    Movies MovieModel
    Tokens TokenModel // Add a new Tokens field.
    Users UserModel
}

func NewModels(db *sql.DB) Models {
    return Models{
        Movies: MovieModel{DB: db},
        Tokens: TokenModel{DB: db}, // Initialize a new TokenModel instance.
        Users:  UserModel{DB: db},
    }
}
```

At this point you should be able to restart the application, and everything should work without a hitch.

```
$ go run ./cmd/api/  
{ "level": "INFO", "time": "2021-04-15T16:07:58Z", "message": "database connection pool established" }  
{ "level": "INFO", "time": "2021-04-15T16:07:58Z", "message": "starting server", "properties": { "addr": ":4000", "env": "development" } }
```

Additional Information

The `math/rand` package

Go also has a `math/rand` package which provides a *deterministic* pseudo-random number generator (PRNG). It's important that you never use the `math/rand` package for any purpose where cryptographic security is required, such as generating tokens or secrets like we are here.

In fact, it's arguably best to use `crypto/rand` as *standard practice*. Only opt for using `math/rand` in specific scenarios where you are certain that a deterministic PRNG is acceptable, and you actively need the *faster performance* of `math/rand`.

Sending Activation Tokens

The next step is to hook this up to our `registerUserHandler`, so that we generate an activation token when a user signs up and include it in their welcome email — similar to this:

```
Hi,  
  
Thanks for signing up for a Greenlight account. We're excited to have you on board!  
  
For future reference, your user ID number is 123.  
  
Please send a request to the `PUT /v1/users/activated` endpoint with the following JSON  
body to activate your account:  
  
{ "token": "Y3QMGX3PJ3WLRL2YRTQGQ6KRHU" }  
  
Please note that this is a one-time use token and it will expire in 3 days.  
  
Thanks,  
  
The Greenlight Team
```

The most important thing about this email is that we're instructing the user to activate by issuing a `PUT` request to our API — not by *clicking a link* which contains the token as part of the URL path or query string.

Having a user click a link to activate via a `GET` request (which is used by default when clicking a link) would certainly be more convenient, but in the case of our API it has some big drawbacks. In particular:

- It would violate the HTTP principle that the `GET` method should only be used for 'safe' requests which retrieve resources — not for requests that modify something (like a user's activation status).
- It's possible that the user's web browser or antivirus will pre-fetch the link URL in the background, inadvertently activating the account. [This Stack Overflow comment](#) explains the risk of this nicely:

That could result in a scenario where a malicious actor (Eve) wants to make an account using someone else's email (Alice). Eve signs up, and Alice received an email. Alice opens the email because she is curious about an account she didn't request. Her browser (or antivirus) requests the URL in the background, inadvertently activating the account.

All-in-all, you should make sure that any actions which change the state of your application (including activating a user) are only ever executed via **POST**, **PUT**, **PATCH** or **DELETE** requests — not by **GET** requests.

Note: If your API is the back-end for a website, then you could adapt this email so that it asks the user to click a link which takes them to a page on your website. They can *then* click a button on the page to ‘confirm their activation’, which performs the **PUT** request to your API that actually activates the user. We’ll look at this pattern in more detail in the next chapter.

But for now, if you’re following along, go ahead and update your welcome email templates to include the activation token as follows:

File: internal/mailer/templates/user_welcome.tmpl

```
{{define "subject"}}Welcome to Greenlight!{{end}}

{{define "plainBody"}}
Hi,

Thanks for signing up for a Greenlight account. We're excited to have you on board!

For future reference, your user ID number is {{.userID}}.

Please send a request to the `PUT /v1/users/activated` endpoint with the following JSON
body to activate your account:

{"token": "{{.activationToken}}"}

Please note that this is a one-time use token and it will expire in 3 days.

Thanks,

The Greenlight Team
{{end}}

{{define "htmlBody"}}
<!doctype html>
<html>

<head>
  <meta name="viewport" content="width=device-width" />
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>

<body>
  <p>Hi,</p>
  <p>Thanks for signing up for a Greenlight account. We're excited to have you on board!</p>
  <p>For future reference, your user ID number is {{.userID}}.</p>
  <p>Please send a request to the <code>PUT /v1/users/activated</code> endpoint with the
  following JSON body to activate your account:</p>
  <pre><code>
{"token": "{{.activationToken}}"}
</code></pre>
  <p>Please note that this is a one-time use token and it will expire in 3 days.</p>
  <p>Thanks,</p>
  <p>The Greenlight Team</p>
</body>

</html>
{{end}}
```

Next we'll need to update the `registerUserHandler` to generate a new activation token, and pass it to the welcome email template as dynamic data, along with the user ID.

Like so:

File: cmd/api/users.go

```
package main

import (
    "errors"
    "net/http"
    "time" // New import

    "greenlight.alexedwards.net/internal/data"
    "greenlight.alexedwards.net/internal/validator"
)

func (app *application) registerUserHandler(w http.ResponseWriter, r *http.Request) {
    ...

    err = app.models.Users.Insert(user)
    if err != nil {
        switch {
        case errors.Is(err, data.ErrDuplicateEmail):
            v.AddError("email", "a user with this email address already exists")
            app.failedValidationResponse(w, r, v.Errors)
        default:
            app.serverErrorResponse(w, r, err)
        }
        return
    }

    // After the user record has been created in the database, generate a new activation
    // token for the user.
    token, err := app.models.Tokens.New(user.ID, 3*24*time.Hour, data.ScopeActivation)
    if err != nil {
        app.serverErrorResponse(w, r, err)
        return
    }

    app.background(func() {
        // As there are now multiple pieces of data that we want to pass to our email
        // templates, we create a map to act as a 'holding structure' for the data. This
        // contains the plaintext version of the activation token for the user, along
        // with their ID.
        data := map[string]interface{}{
            "activationToken": token.Plaintext,
            "userID":          user.ID,
        }

        // Send the welcome email, passing in the map above as dynamic data.
        err = app.mailer.Send(user.Email, "user_welcome.tmpl", data)
        if err != nil {
            app.logger.PrintError(err, nil)
        }
    })

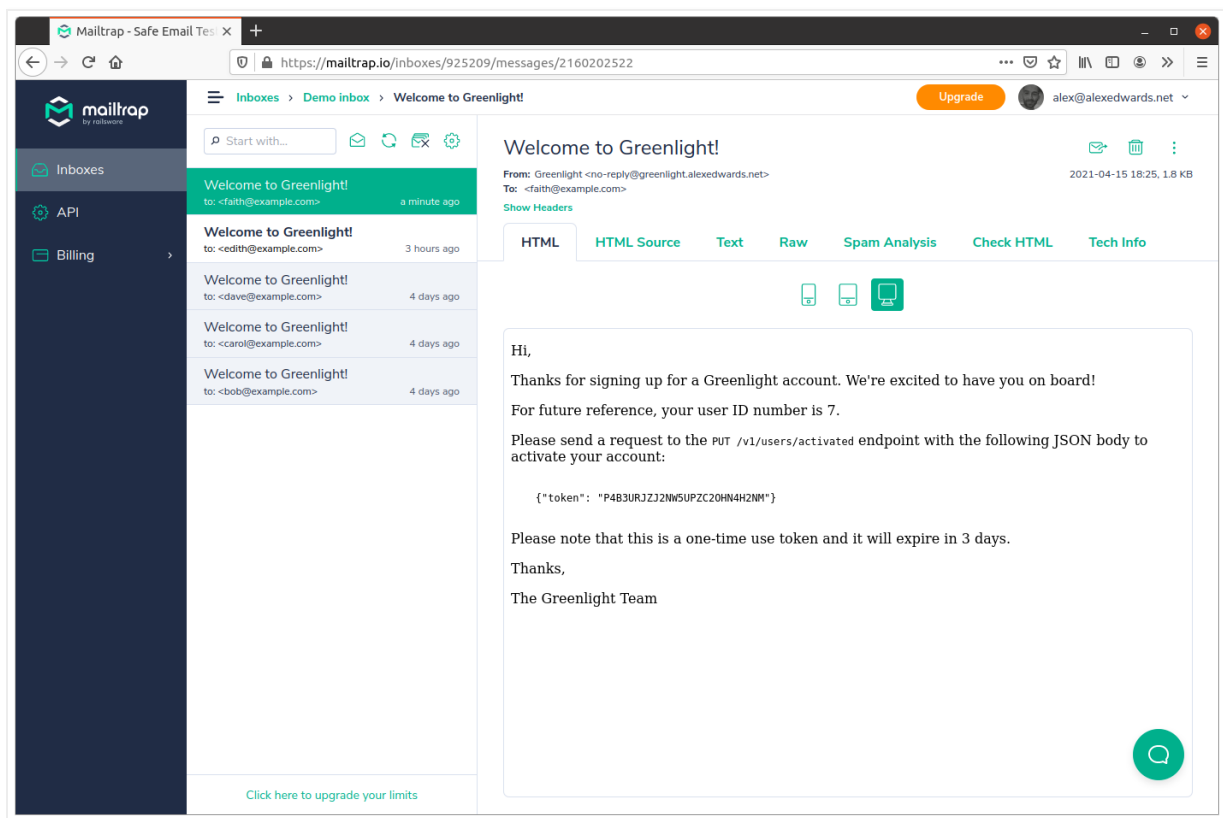
    err = app.writeJSON(w, http.StatusOK, envelope{"user": user}, nil)
    if err != nil {
        app.serverErrorResponse(w, r, err)
    }
}
```

OK, let's see how this works...

Restart the application, and then register a new user account with the email address **faith@example.com**. Similar to this:

```
$ BODY='{"name": "Faith Smith", "email": "faith@example.com", "password": "pa55word"}'
$ curl -d "$BODY" localhost:4000/v1/users
{
  "user": {
    "id": 7,
    "created_at": "2021-04-15T20:25:41+02:00",
    "name": "Faith Smith",
    "email": "faith@example.com",
    "activated": false
  }
}
```

And if you open your Mailtrap inbox again, you should now see the new welcome email containing the activation token for **faith@example.com**, like so:



So, in my case, we can see that **faith@example.com** has been sent the activation token **P4B3URJZJ2NW5UPZC20HN4H2NM**. If you're following along, your token should be a different 26-character string.

Out of interest, let's quickly look inside the **tokens** table in our PostgreSQL database. Again, the exact values in your database will be different, but it should look similar to this:

```
$ psql $GREENLIGHT_DB_DSN
Password for user greenlight:
psql (13.0 (Ubuntu 13.0-1.pgdg20.04+1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression: off)
Type "help" for help.

greenlight=> SELECT * FROM tokens;
      hash                                     | user_id | expiry           | scope
-----+-----+-----+-----
 \x09bcb40206b25fe511bfef4d56cbe8c4a141869fc29612fa984b371ef086f5f5 |      7 | 2021-04-18 20:25:41+02 | activation
```

We can see here that the activation token hash is displayed as the value:

```
09bcb40206b25fe511bfef4d56cbe8c4a141869fc29612fa984b371ef086f5f5
```

As we mentioned earlier, `psql` always displays values in `bytea` columns as a hex-encoded string. So what we're seeing here is a *hexadecimal encoding of the SHA-256 hash of the plaintext token* `P4B3URJZJ2NW5UPZC20HN4H2NM` that we sent in the welcome email.

Note: If you want, you can verify that this is correct by entering a plaintext activation token from the welcome email into this [online SHA-256 hash generator](#). You should see that the result matches the hex-encoded value inside your PostgreSQL database.

Notice too that the `user_id` value of `7` is correct for our `faith@example.com` user, the expiry time has been correctly set to three days from now, and the token has the scope value `activation`?

Additional Information

A standalone endpoint for generating tokens

You may also want to provide a standalone endpoint for generating and sending activation tokens to your users. This can be useful if you need to re-send an activation token, such as when a user doesn't activate their account within the 3-day time limit, or they never receive their welcome email.

The code to implement this endpoint is a mix of patterns that we've talked about already, so rather than repeating them in the main flow of the book the instructions are included in [this appendix](#).

Activating a User

In this chapter we're going to move on to the part of the activation workflow where we actually activate a user. But before we write any code, I'd like to quickly talk about the relationship between users and tokens in our system.

What we have is known in relational database terms as a *one-to-many relationship* — where one user may have many tokens, but a token can only belong to one user.

When you have a one-to-many relationship like this, you'll potentially want to execute queries against the relationship from two different sides. In our case, for example, we might want to either:

- Retrieve the user associated with a token.
- Retrieve all tokens associated with a user.

To implement these queries in your code, a clean and clear approach is to update your database models to include some additional methods like this:

```
UserModel.GetForToken(token) → Retrieve the user associated with a token  
TokenModel.GetAllForUser(user) → Retrieve all tokens associated with a user
```

The nice thing about this approach is that the entities being returned align with the main responsibility of the models: the `UserModel` method is returning a user, and the `TokenModel` method is returning tokens.

Creating the activateUserHandler

Now that we've got a very high-level idea of how we're going to query the user ↔ token relationship in our database models, let's start to build up the code for activating a user.

In order to do this, we'll need to add a new `PUT /v1/users/activated` endpoint to our API:

Method	URL Pattern	Handler	Action
GET	/v1/healthcheck	healthcheckHandler	Show application information
GET	/v1/movies	listMoviesHandler	Show the details of all movies
POST	/v1/movies	createMovieHandler	Create a new movie
GET	/v1/movies/:id	showMovieHandler	Show the details of a specific movie
PATCH	/v1/movies/:id	updateMovieHandler	Update the details of a specific movie
DELETE	/v1/movies/:id	deleteMovieHandler	Delete a specific movie
POST	/v1/users	registerUserHandler	Register a new user
PUT	/v1/users/activated	activateUserHandler	Activate a specific user

And the workflow will look like this:

1. The user submits the plaintext activation token (which they just received in their email) to the **PUT /v1/users/activated** endpoint.
2. We validate the plaintext token to check that it matches the expected format, sending the client an error message if necessary.
3. We then call the **UserModel.GetForToken()** method to retrieve the details of the user associated with the provided token. If there is no matching token found, or it has expired, we send the client an error message.
4. We activate the associated user by setting **activated = true** on the user record and update it in our database.
5. We delete all activation tokens for the user from the **tokens** table. We can do this using the **TokenModel.DeleteAllForUser()** method that we made earlier.
6. We send the updated user details in a JSON response.

Let's begin in our **cmd/api/users.go** file and create the new **activateUserHandler** to work through these steps:

File: cmd/api/users.go

```
package main

...

func (app *application) activateUserHandler(w http.ResponseWriter, r *http.Request) {
    // Parse the plaintext activation token from the request body.
    var input struct {
        TokenPlaintext string `json:"token"`
    }
}
```

```

err := app.readJSON(w, r, &input)
if err != nil {
    app.badRequestResponse(w, r, err)
    return
}

// Validate the plaintext token provided by the client.
v := validator.New()

if data.ValidateTokenPlaintext(v, input.TokenPlaintext); !v.Valid() {
    app.failedValidationResponse(w, r, v.Errors)
    return
}

// Retrieve the details of the user associated with the token using the
// GetForToken() method (which we will create in a minute). If no matching record
// is found, then we let the client know that the token they provided is not valid.
user, err := app.models.Users.GetForToken(data.ScopeActivation, input.TokenPlaintext)
if err != nil {
    switch {
    case errors.Is(err, data.ErrRecordNotFound):
        v.AddError("token", "invalid or expired activation token")
        app.failedValidationResponse(w, r, v.Errors)
    default:
        app.serverErrorResponse(w, r, err)
    }
    return
}

// Update the user's activation status.
user.Activated = true

// Save the updated user record in our database, checking for any edit conflicts in
// the same way that we did for our movie records.
err = app.models.Users.Update(user)
if err != nil {
    switch {
    case errors.Is(err, data.ErrEditConflict):
        app.editConflictResponse(w, r)
    default:
        app.serverErrorResponse(w, r, err)
    }
    return
}

// If everything went successfully, then we delete all activation tokens for the
// user.
err = app.models.Tokens.DeleteAllForUser(data.ScopeActivation, user.ID)
if err != nil {
    app.serverErrorResponse(w, r, err)
    return
}

// Send the updated user details to the client in a JSON response.
err = app.writeJSON(w, http.StatusOK, envelope{"user": user}, nil)
if err != nil {
    app.serverErrorResponse(w, r, err)
}
}

```

If you try to compile the application at this point, you'll get an error because the `UserModel.GetForToken()` method doesn't yet exist. Let's go ahead and create that now.

The UserModel.GetForToken method

As we mentioned above, we want the `UserModel.GetForToken()` method to retrieve the details of the user associated with a particular activation token. If there is no matching token found, or it has expired, we want this to return a `ErrRecordNotFound` error instead.

In order to do that, we'll need to execute the following SQL query on our database:

```
SELECT users.id, users.created_at, users.name, users.email, users.password_hash, users.activated, users.version
FROM users
INNER JOIN tokens
ON users.id = tokens.user_id
WHERE tokens.hash = $1
AND tokens.scope = $2
AND tokens.expiry > $3
```

This is more complicated than most of the SQL queries we've used so far, so let's take a moment to explain what it is doing.

In this query we are using `INNER JOIN` to join together information from the `users` and `tokens` tables. Specifically, we're using the `ON users.id = tokens.user_id` clause to indicate that we want to join records *where the user id value equals the token user_id*.

Behind the scenes, you can think of `INNER JOIN` as creating an 'interim' table containing the joined data from *both* tables. Then, in our SQL query, we use the `WHERE` clause to filter this interim table to leave only rows where the token hash and token scope match specific placeholder parameter values, and the token expiry is after a specific time. Because the token hash is also a primary key, we will always be left with exactly one record which contains the details of the user associated with the token hash (or no records at all, if there wasn't a matching token).

Hint: If you're not familiar with performing joins in SQL, then [this article](#) provides a good overview of the different types of joins, how they work, and some examples that should help assist your understanding.

If you're following along, open up your `internal/data/users.go` file and add a `GetForToken()` method which executes this SQL query like so:

```
File: internal/data/users.go
```

```
package data
```

```
import (
```

```

"context"
"crypto/sha256" // New import
"database/sql"
"errors"
"time"

"greenlight.alexedwards.net/internal/validator"

"golang.org/x/crypto/bcrypt"
)

...

func (m UserModel) GetForToken(tokenScope, tokenPlaintext string) (*User, error) {
    // Calculate the SHA-256 hash of the plaintext token provided by the client.
    // Remember that this returns a byte *array* with length 32, not a slice.
    tokenHash := sha256.Sum256([]byte(tokenPlaintext))

    // Set up the SQL query.
    query := `
        SELECT users.id, users.created_at, users.name, users.email, users.password_hash, users.activated, users.version
        FROM users
        INNER JOIN tokens
        ON users.id = tokens.user_id
        WHERE tokens.hash = $1
        AND tokens.scope = $2
        AND tokens.expiry > $3`

    // Create a slice containing the query arguments. Notice how we use the [:] operator
    // to get a slice containing the token hash, rather than passing in the array (which
    // is not supported by the pq driver), and that we pass the current time as the
    // value to check against the token expiry.
    args := []interface{}{tokenHash[:], tokenScope, time.Now()}

    var user User

    ctx, cancel := context.WithTimeout(context.Background(), 3*time.Second)
    defer cancel()

    // Execute the query, scanning the return values into a User struct. If no matching
    // record is found we return an ErrRecordNotFound error.
    err := m.DB.QueryRowContext(ctx, query, args...).Scan(
        &user.ID,
        &user.CreatedAt,
        &user.Name,
        &user.Email,
        &user.Password.hash,
        &user.Activated,
        &user.Version,
    )
    if err != nil {
        switch {
        case errors.Is(err, sql.ErrNoRows):
            return nil, ErrRecordNotFound
        default:
            return nil, err
        }
    }

    // Return the matching user.
    return &user, nil
}

```

Now that is in place, the final thing we need to do is add the **PUT /v1/users/activated**

endpoint to the `cmd/api/routes.go` file. Like so:

```
File: cmd/api/routes.go

package main

...

func (app *application) routes() http.Handler {
    router := httprouter.New()

    router.NotFound = http.HandlerFunc(app.notFoundResponse)
    router.MethodNotAllowed = http.HandlerFunc(app.methodNotAllowedResponse)

    router.HandleFunc(http.MethodGet, "/v1/healthcheck", app.healthcheckHandler)

    router.HandleFunc(http.MethodGet, "/v1/movies", app.listMoviesHandler)
    router.HandleFunc(http.MethodPost, "/v1/movies", app.createMovieHandler)
    router.HandleFunc(http.MethodGet, "/v1/movies/:id", app.showMovieHandler)
    router.HandleFunc(http.MethodPatch, "/v1/movies/:id", app.updateMovieHandler)
    router.HandleFunc(http.MethodDelete, "/v1/movies/:id", app.deleteMovieHandler)

    router.HandleFunc(http.MethodPost, "/v1/users", app.registerUserHandler)
    // Add the route for the PUT /v1/users/activated endpoint.
    router.HandleFunc(http.MethodPut, "/v1/users/activated", app.activateUserHandler)

    return app.recoverPanic(app.rateLimit(router))
}
```

As an aside, I should quickly explain that the reason we're using **PUT** rather than **POST** for this endpoint is because it's *idempotent*.

If a client sends the same **PUT** `/v1/users/activated` request multiple times, the first will succeed (assuming the token is valid) and then any subsequent requests will result in an error being sent to the client (because the token has been used and deleted from the database). But the important thing is that *nothing in our application state (i.e. database) changes after that first request*.

Basically, there are no application state side-effects from the client sending the same request multiple times, which means that the endpoint is idempotent and using **PUT** is more appropriate than **POST**.

Alright, let's restart the API and then try this out.

First, try making some requests to the **PUT** `/v1/users/activated` endpoint containing some invalid tokens. You should get the appropriate error messages in response, like so:

```
$ curl -X PUT -d '{"token": "invalid"}' localhost:4000/v1/users/activated
{
  "error": {
    "token": "must be 26 bytes long"
  }
}

$ curl -X PUT -d '{"token": "ABCDEFGHIJKLMNOPQRSTUVWXYZ"}' localhost:4000/v1/users/activated
{
  "error": {
    "token": "invalid or expired activation token"
  }
}
```

Then try making a request using a valid activation token from one of your emails (which will be in your Mailtrap inbox if you're following along). In my case, I'll use the token **P4B3URJZJ2NW5UPZC20HN4H2NM** to activate the user **faith@example.com** (who we created in the previous chapter).

You should get a JSON response back with an **activated** field that confirms that the user has been activated, similar to this:

```
$ curl -X PUT -d '{"token": "P4B3URJZJ2NW5UPZC20HN4H2NM"}' localhost:4000/v1/users/activated
{
  "user": {
    "id": 7,
    "created_at": "2021-04-15T20:25:41+02:00",
    "name": "Faith Smith",
    "email": "faith@example.com",
    "activated": true
  }
}
```

And if you try repeating the request again with the same token, you should now get an **"invalid or expired activation token"** error due to the fact we have deleted all activation tokens for **faith@example.com**.

```
$ curl -X PUT -d '{"token": "P4B3URJZJ2NW5UPZC20HN4H2NM"}' localhost:4000/v1/users/activated
{
  "error": {
    "token": "invalid or expired activation token"
  }
}
```

Important: In production, with activation tokens for real accounts, you must make sure that the tokens are only ever accepted over an encrypted HTTPS connection — not via regular HTTP like we are using here.

Lastly, let's take a quick look in our database to see the state of our **users** table.

```
$ psql $GREENLIGHT_DB_DSN
Password for user greenlight:
psql (13.1 (Ubuntu 13.1-1.pgdg20.04+1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression: off)
Type "help" for help.

greenlight=> SELECT email, activated, version FROM users;
 email      | activated | version
-----+-----+-----
alice@example.com | f       | 1
bob@example.com  | f       | 1
carol@example.com | f       | 1
dave@example.com | f       | 1
edith@example.com | f       | 1
faith@example.com | t       | 2
```

In contrast to all the other users, we can see that **faith@example.com** has now got the value **activated = true** and the version number for their user record has been bumped up to **2**.

Additional Information

Web application workflow

If your API is the backend to a website, rather than a completely standalone service, you can tweak the activation workflow to make it simpler and more intuitive for users while still being secure.

There are two main options here. The first, and most robust, option is to ask the user to copy-and-paste the token into a form on your website which then performs the **PUT /v1/users/activate** request for them using some JavaScript. The welcome email to support that workflow could look something like this:

Hi,

Thanks for signing up for a Greenlight account. We're excited to have you on board!

For future reference, your user ID number is 123.

To activate your Greenlight account please visit <https://example.com/users/activate> and enter the following code:

Y3QMGX3PJ3WLRL2YRTQGQ6KRHU

Please note that this code will expire in 3 days and can only be used once.

Thanks,

The Greenlight Team

This approach is fundamentally simple and secure — effectively your website just provides a form that performs the **PUT** request for the user, rather than them needing to do it manually using **curl** or another tool.

Note: When creating the link in this email, don't rely on the **Host** header from **r.Host** to construct the URL, as that would be vulnerable to a **host header injection attack**. The URL domain should be either be hard-coded or passed in as a command-line flag when starting the application.

Alternatively, if you don't want the user to copy-and-paste a token, you could ask them to click a link containing the token which takes them to a page on your website. Similar to this:

Hi,

Thanks for signing up for a Greenlight account. We're excited to have you on board!

For future reference, your user ID number is 123.

To activate your Greenlight account please click the following link:

<https://example.com/users/activate?token=Y3QMGX3PJ3WLRL2YRTQGQ6KRHU>

Please note that this link will expire in 3 days and can only be used once.

Thanks,

The Greenlight Team

This page should then display a button that says something like 'Confirm your account activation', and some JavaScript on the webpage can extract the token from the URL and submit it to your **PUT /v1/users/activate** API endpoint when the user clicks the button.

If you go with this second option, you also need to take steps to avoid the token being [leaked in a referrer header](#) if the user navigates to a different site. You can use the [Referrer-Policy: Origin](#) header or `<meta name="referrer" content="origin">` HTML tag to mitigate this, although you should be aware that it's not supported by absolutely all web browsers (support is [currently at ~96%](#)).

In all cases though, whatever the email and workflow looks like in terms of the front-end and user-experience, the back-end API endpoint that we've implemented is the same and doesn't need to change.

SQL query timing attack

It's worth pointing out that the SQL query we're using in `UserModel.GetForToken()` is theoretically vulnerable to a timing attack, because PostgreSQL's evaluation of the `tokens.hash = $1` condition is not performed in constant-time.

```
SELECT users.id, users.created_at, users.name, users.email, users.password_hash, users.activated, users.version
FROM users
INNER JOIN tokens
ON users.id = tokens.user_id
WHERE tokens.hash = $1    --<-- This is vulnerable to a timing attack
AND tokens.scope = $2
AND tokens.expiry > $3
```

Although it would be somewhat tricky to pull off, in theory an attacker could issue thousands of requests to our `PUT /v1/users/activated` endpoint and analyze tiny discrepancies in the average response time to build up a picture of a hashed activation token value in the database.

But, in our case, even if a timing attack was successful it would only leak the *hashed* token value from the database — not the plaintext token value that the user actually needs to submit to activate their account.

So the attacker would still need to use brute-force to find a 26-character string which happens to have the same SHA-256 hash that they discovered from the timing attack. This is incredibly difficult to do, and simply not viable with current technology.

Authentication

In this section of the book we're going to look at how to *authenticate* requests to our API, so that we know exactly which user a particular request is coming from.

Remember: *Authentication* is about confirming *who a user is*, whereas *authorization* is about checking whether that user is permitted to do something.

We will:

- Lay out the possible approaches to API authentication that we could use, and talk through their relative pros and cons.
- Implement a stateful token-based authentication pattern, which allows clients to exchange their user credentials for a time-limited authentication token identifying who they are.

Authentication Options

Before we get started on writing any code, let's talk about *how* we're going to authenticate requests to our API and find out which user a request is coming from.

Choosing a high-level approach to API authentication can be tricky — there are many different options, and it's not always immediately clear which one is the right fit for your project. So in this chapter we'll discuss some of the most common approaches at a high level, talk through their relative pros and cons, and finish with some general guidelines about when they're appropriate to use.

Specifically, the five approaches that we'll compare are:

- Basic authentication
- Stateful token authentication
- Stateless token authentication
- API key authentication
- OAuth 2.0 / OpenID Connect

Important: For all authentication methods that we're describing in this chapter, it's assumed that your API only communicates with clients over HTTPS.

HTTP basic authentication

Perhaps the simplest way to determine *who is making a request* to your API is to use HTTP basic authentication.

With this method, the client includes an **Authorization** header with every request containing their credentials. The credentials need to be in the format **username:password** and base-64 encoded. So, for example, to authenticate as **alice@example.com:pa55word** the client would send the following header:

```
Authorization: Basic YWxpY2VAZXhhbXBsZS5jb206cGE1NXdvcnQ=
```

In your API, you can then extract the credentials from this header using Go's

`Request.BasicAuth()` method, and verify that they're correct before continuing to process the request.

A big plus of HTTP basic authentication is how simple it is for clients. They can just send the same header with every request — and HTTP basic authentication is supported out-of-the-box by most programming languages, web browsers, and tools such as `curl` and `wget`.

It's often useful in the scenario where your API doesn't have 'real' user accounts, but you want a quick and easy way to restrict access to it or protect it from prying eyes.

For APIs with 'real' user accounts and — in particular — hashed passwords, it's not such a great fit. Comparing the password provided by a client against a (slow) hashed password is a deliberately costly operation, and when using HTTP basic authentication you need to do that check for every request. That will create a lot of extra work for your API server and add significant latency to responses.

But even then, basic authentication can still be a good choice if traffic to your API is very low and response speed is not important to you.

Token authentication

The high-level idea behind *token authentication* (also sometimes known as *bearer token authentication*) works like this:

1. The client sends a request to your API containing their credentials (typically username or email address, and password).
2. The API verifies that the credentials are correct, generates a *bearer token* which represents the user, and sends it back to the user. The token expires after a set period of time, after which the user will need to resubmit their credentials again to get a new token.
3. For subsequent requests to the API, the client includes the token in an **Authorization** header like this:

```
Authorization: Bearer <token>
```

4. When your API receives this request, it checks that the token hasn't expired and examines the token value to determine who the user is.

For APIs where user passwords are hashed (like ours), this approach is better than basic

authentication because it means that the slow password check only has to be done periodically — either when creating a token for the first time or after a token has expired.

The downside is that managing tokens can be complicated for clients — they will need to implement the necessary logic for caching tokens, monitoring and managing token expiry, and periodically generating new tokens.

We can break down token authentication further into two sub-types: *stateful* and *stateless* token authentication. They are quite different in terms of their pros and cons, so let's discuss them both separately.

Stateful token authentication

In a stateful token approach, the value of the token is a high-entropy cryptographically-secure random string. This token — or a fast hash of it — is stored server-side in a database, alongside the user ID and an expiry time for the token.

When the client sends back the token in subsequent requests, your API can look up the token in the database, check that it hasn't expired, and retrieve the corresponding user ID to find out who the request is coming from.

The big advantage of this is that your API maintains control over the tokens — it's straightforward to revoke tokens on a per-token or per-user basis by deleting them from the database or marking them as expired.

Conceptually it's also simple and robust — the security is provided by the token being an 'unguessable', which is why it's important to use a high-entropy cryptographically-secure random value for the token.

So, what are the downsides?

Beyond the complexity for clients that is inherent with token authentication generally, it's difficult to find much to criticize about this approach. Perhaps the fact that it requires a database lookup is a negative — but in most cases you will need to make a database lookup to check the user's activation status or retrieve additional information about them *anyway*.

Stateless token authentication

In contrast, stateless tokens encode the user ID and expiry time *in the token itself*. The token is cryptographically signed to prevent tampering and (in some cases) encrypted to prevent the contents being read.

There are a few different technologies that you can use to create stateless tokens. Encoding

the information in a [JWT](#) (JSON Web Token) is probably the most well-known approach, but [PASETO](#), [Branca](#) and [nacl/secretbox](#) are viable alternatives too. Although the implementation details of these technologies are different, the overarching pros and cons in terms of authentication are similar.

The main selling point of using stateless tokens for authentication is that the work to encode and decode the token can be done in memory, and all the information required to identify the user is contained within the token itself. There's no need to perform a database lookup to find out who a request is coming from.

The primary downside of stateless tokens is that they can't easily be revoked once they are issued.

In an emergency, you could effectively revoke *all* tokens by changing the secret used for signing your tokens (forcing all users to re-authenticate), or another workaround is to maintain a blocklist of revoked tokens in a database (although that defeats the 'stateless' aspect of having stateless tokens).

Note: You should generally avoid storing additional information in a stateless token, such as a user's activation status or permissions, and using that as the basis for *authorization* checks. During the lifetime of the token, the information encoded into it will potentially become stale and out-of-sync with the real data in your system — and relying on stale data for authorization checks can easily lead to unexpected behavior for users and various security issues.

Finally, with JWTs in particular, the fact that they're highly configurable means that there are *lots of things you can get wrong*. The [Critical vulnerabilities in JSON Web Token libraries](#) and [JWT Security Best Practices](#) articles provide a good introduction to the type of things you need to be careful of here.

Because of these downsides, stateless tokens — and JWTs in particular — are generally not the best choice for managing authentication in most API applications.

But they *can* be very useful in a scenario where you need *delegated authentication* — where the application *creating* the authentication token is different to the application *consuming* it, and those applications don't share any state (which means that using stateful tokens isn't an option). For instance, if you're building a system which has a microservice-style architecture behind the scenes, then a stateless token created by an 'authentication' service can subsequently be passed to other services to identify the user.

API-key authentication

The idea behind API-key authentication is that a user has a non-expiring secret ‘key’ associated with their account. This key should be a high-entropy cryptographically-secure random string, and a fast hash of the key (SHA256 or SHA512) should be stored alongside the corresponding user ID in your database.

The user then passes their key with each request to your API in a header like this:

```
Authorization: Key <key>
```

On receiving it, your API can regenerate the fast hash of the key and use it to lookup the corresponding user ID from your database.

Conceptually, this isn’t a million miles away from the stateful token approach — the main difference is that the keys are permanent keys, rather than temporary tokens.

On one hand, this is nice for the client as they can use the same key for every request and they don’t need to write code to manage tokens or expiry. On the other hand, the user now has two long-lived secrets to manage which can potentially compromise their account: their password, and their API key.

Supporting API keys also adds additional complexity to your API application — you’ll need a way for users to regenerate their API key if they lose it or the key is compromised, and you may also wish to support multiple API keys for the same user, so they can use different keys for different purposes.

It’s also important to note that API keys themselves should only ever be communicated to users over a secure channel, and you should treat them with the same level of care that you would a user’s password.

OAuth 2.0 / OpenID Connect

Another option is to leverage OAuth 2.0 for authentication. With this approach, information about your users (and their passwords) is stored by a third-party *identity provider* like Google or Facebook rather than yourself.

The first thing to mention here is that *OAuth 2.0 is not an authentication protocol*, and you shouldn’t really use it for authenticating users. The oauth.net website has a great article [explaining this](#), and I highly recommend reading it.

If you want to implement authentication checks against a third-party identity provider, you should use [OpenID Connect](#) (which is built directly on top of OAuth 2.0).

There's a comprehensive overview of OpenID Connect [here](#), but at a very, very, high level it works like this:

- When you want to authenticate a request, you redirect the user to an 'authentication and consent' form hosted by the identity provider.
- If the user consents, then the identity provider sends your API an *authorization code*.
- Your API then sends the authorization code to another endpoint provided by the identity provider. They verify the authorization code, and if it's valid they will send you a JSON response containing an *ID token*.
- This ID token is itself a JWT. You need to validate and decode this JWT to get the actual user information, which includes things like their email address, name, birth date, timezone etc.
- Now that you know who the user is, you can then implement a stateful or stateless authentication token pattern so that you don't have to go through the whole process for every subsequent request.

Like all the other options we've looked at, there are pros and cons to using OpenID Connect. The big plus is that you don't need to persistently store user information or passwords yourself. The big downside is that it's quite complex — although there are some helper packages like [coreos/go-oidc](#) which do a good job of masking that complexity and providing a simple interface for the OpenID Connect workflow that you can hook in to.

It's also important to point out that using OpenID Connect requires all your users to have an account with the identity provider, and the 'authentication and consent' step requires human interaction via a web browser — which is probably fine if your API is the back-end for a website, but not ideal if it is a 'standalone' API with other computer programs as clients.

What authentication approach should I use?

It's difficult to give blanket guidance on what authentication approach is best to use for your API. As with most things in programming, different tools are appropriate for different jobs.

But as simple, rough, rules-of-thumb:

- If your API doesn't have 'real' user accounts with slow password hashes, then HTTP basic authentication can be a good — and often overlooked — fit.

- If you don't want to store user passwords yourself, all your users have accounts with a third-party identity provider that supports OpenID Connect, and your API is the back-end for a website... then use OpenID Connect.
- If you require delegated authentication, such as when your API has a microservice architecture with different services for performing authentication and performing other tasks, then use stateless authentication tokens.
- Otherwise use API keys or stateful authentication tokens. In general:
 - Stateful authentication tokens are a nice fit for APIs that act as the back-end for a website or single-page application, as there is a natural moment when the user logs-in where they can be exchanged for user credentials.
 - In contrast, API keys can be better for more 'general purpose' APIs because they're permanent and simpler for developers to use in their applications and scripts.

In the rest of this book, we're going to implement authentication using the *stateful authentication token* pattern. In our case we've already built a lot of the necessary logic for this as part of our *activation tokens* work.

Note: Although I really don't recommend using JWTs unless you need some form of *delegated authentication*, I'm aware that they hold a lot of mind-share in the developer community and are often used more widely than that.

So, because of this, I've also included [an appendix](#) at the end of the book which explains how the *Greenlight* API can be adapted to use a stateless authentication token pattern with JWTs.

Generating Authentication Tokens

In this chapter we're going to focus on building up the code for a new `POST/v1/tokens/authentication` endpoint, which will allow a client to exchange their credentials (email address and password) for a stateful authentication token.

Note: For conciseness, rather than repeating the words 'stateful authentication token' throughout the rest of this book, from now on we'll simply refer this as the user's *authentication token*.

At a high level, the process for exchanging a user's credentials for an authentication token will work like this:

1. The client sends a JSON request to a new `POST/v1/tokens/authentication` endpoint containing their credentials (email and password).
2. We look up the user record based on the email, and check if the password provided is the correct one for the user. If it's not, then we send an error response.
3. If the password is correct, we use our `app.models.Tokens.New()` method to generate a token with an expiry time of 24 hours and the scope `"authentication"`.
4. We send this authentication token back to the client in a JSON response body.

Let's begin in our `internal/data/tokens.go` file.

We need to update this file to define a new `"authentication"` scope, and add some struct tags to customize how the `Token` struct appears when it is encoded to JSON. Like so:

File: internal/data/tokens.go

```
package data

...

const (
    ScopeActivation    = "activation"
    ScopeAuthentication = "authentication" // Include a new authentication scope.
)

// Add struct tags to control how the struct appears when encoded to JSON.
type Token struct {
    Plaintext string `json:"token"`
    Hash      []byte   `json:"- "`
    UserID    int64    `json:"- "`
    Expiry    time.Time `json:"expiry"`
    Scope     string   `json:"- "`
}

...
```

These new struct tags mean that only the `Plaintext` and `Expiry` fields will be included when encoding a `Token` struct — all the other fields will be omitted. We also rename the `Plaintext` field to `"token"`, just because it's a more meaningful name for clients than 'plaintext' is.

Altogether, this means that when we encode a `Token` struct to JSON the result will look similar to this:

```
{
  "token": "X3ASTT2CDAN66BACKSCI4SU7SI",
  "expiry": "2021-01-18T13:00:25.648511827+01:00"
}
```

Building the endpoint

Now let's get into the meat of this chapter and set up all the code for the new `POST/v1/tokens/authentication` endpoint. By the time we're finished, our API routes will look like this:

Method	URL Pattern	Handler	Action
GET	/v1/healthcheck	healthcheckHandler	Show application information
GET	/v1/movies	listMoviesHandler	Show the details of all movies
POST	/v1/movies	createMovieHandler	Create a new movie
GET	/v1/movies/:id	showMovieHandler	Show the details of a specific movie
PATCH	/v1/movies/:id	updateMovieHandler	Update the details of a specific movie
DELETE	/v1/movies/:id	deleteMovieHandler	Delete a specific movie
POST	/v1/users	registerUserHandler	Register a new user
PUT	/v1/users/activated	activateUserHandler	Activate a specific user
POST	/v1/tokens/authentication	createAuthenticationTokenHandler	Generate a new authentication token

If you're following along, go ahead and create a new `cmd/api/tokens.go` file:

```
$ touch cmd/api/tokens.go
```

And in this new file we'll add the code for the `createAuthenticationTokenHandler`.

Essentially, we want this handler to *exchange the user's email address and password for an authentication token*, like so:

File: `cmd/api/tokens.go`

```
package main

import (
    "errors"
    "net/http"
    "time"

    "greenlight.alexedwards.net/internal/data"
    "greenlight.alexedwards.net/internal/validator"
)

func (app *application) createAuthenticationTokenHandler(w http.ResponseWriter, r *http.Request) {
    // Parse the email and password from the request body.
    var input struct {

```

```

var input struct {
    Email    string `json:"email"`
    Password string `json:"password"`
}

err := app.readJSON(w, r, &input)
if err != nil {
    app.badRequestResponse(w, r, err)
    return
}

// Validate the email and password provided by the client.
v := validator.New()

data.ValidateEmail(v, input.Email)
data.ValidatePasswordPlaintext(v, input.Password)

if !v.Valid() {
    app.failedValidationResponse(w, r, v.Errors)
    return
}

// Lookup the user record based on the email address. If no matching user was
// found, then we call the app.invalidCredentialsResponse() helper to send a 401
// Unauthorized response to the client (we will create this helper in a moment).
user, err := app.models.Users.GetByEmail(input.Email)
if err != nil {
    switch {
    case errors.Is(err, data.ErrRecordNotFound):
        app.invalidCredentialsResponse(w, r)
    default:
        app.serverErrorResponse(w, r, err)
    }
    return
}

// Check if the provided password matches the actual password for the user.
match, err := user.Password.Matches(input.Password)
if err != nil {
    app.serverErrorResponse(w, r, err)
    return
}

// If the passwords don't match, then we call the app.invalidCredentialsResponse()
// helper again and return.
if !match {
    app.invalidCredentialsResponse(w, r)
    return
}

// Otherwise, if the password is correct, we generate a new token with a 24-hour
// expiry time and the scope 'authentication'.
token, err := app.models.Tokens.New(user.ID, 24*time.Hour, data.ScopeAuthentication)
if err != nil {
    app.serverErrorResponse(w, r, err)
    return
}

// Encode the token to JSON and send it in the response along with a 201 Created
// status code.
err = app.writeJSON(w, http.StatusCreated, envelope{"authentication_token": token}, nil)
if err != nil {
    app.serverErrorResponse(w, r, err)
}
}

```

Let's quickly create the `invalidCredentialsResponse()` helper in our `cmd/api/errors.go` file too:

```
File: cmd/api/errors.go

package main

...

func (app *application) invalidCredentialsResponse(w http.ResponseWriter, r *http.Request) {
    message := "invalid authentication credentials"
    app.errorResponse(w, r, http.StatusUnauthorized, message)
}
```

Then lastly, we need to include the `POST /v1/tokens/authentication` endpoint in our application routes. Like so:

```
File: cmd/api/routes.go

package main

...

func (app *application) routes() http.Handler {
    router := httprouter.New()

    router.NotFound = http.HandlerFunc(app.notFoundResponse)
    router.MethodNotAllowed = http.HandlerFunc(app.methodNotAllowedResponse)

    router.HandleFunc(http.MethodGet, "/v1/healthcheck", app.healthcheckHandler)

    router.HandleFunc(http.MethodGet, "/v1/movies", app.listMoviesHandler)
    router.HandleFunc(http.MethodPost, "/v1/movies", app.createMovieHandler)
    router.HandleFunc(http.MethodGet, "/v1/movies/:id", app.showMovieHandler)
    router.HandleFunc(http.MethodPatch, "/v1/movies/:id", app.updateMovieHandler)
    router.HandleFunc(http.MethodDelete, "/v1/movies/:id", app.deleteMovieHandler)

    router.HandleFunc(http.MethodPost, "/v1/users", app.registerUserHandler)
    router.HandleFunc(http.MethodPut, "/v1/users/activated", app.activateUserHandler)

    // Add the route for the POST /v1/tokens/authentication endpoint.
    router.HandleFunc(http.MethodPost, "/v1/tokens/authentication", app.createAuthenticationTokenHandler)

    return app.recoverPanic(app.rateLimit(router))
}
```

With all that complete, we should now be able to generate an authentication token.

Go ahead and make a request to the new `POST /v1/tokens/authentication` endpoint with a valid email address and password for one of the users that you've previously created. You should get a `201 Created` response and a JSON body containing an authentication token, similar to this:

```
$ BODY='{"email": "alice@example.com", "password": "pa55word"}'
$ curl -i -d "$BODY" localhost:4000/v1/tokens/authentication
HTTP/1.1 201 Created
Content-Type: application/json
Date: Fri, 16 Apr 2021 09:03:36 GMT
Content-Length: 125

{
  "authentication_token": {
    "token": "IEYZQUBEMPPAKPOAWTPV6YJ6RM",
    "expiry": "2021-04-17T11:03:36.767078518+02:00"
  }
}
```

In contrast, if you try making a request with a well-formed but unknown email address, or an incorrect password, you should get an error response. For example:

```
$ BODY='{"email": "alice@example.com", "password": "wrong pa55word"}'
$ curl -i -d "$BODY" localhost:4000/v1/tokens/authentication
HTTP/1.1 401 Unauthorized
Content-Type: application/json
Date: Fri, 16 Apr 2021 09:54:01 GMT
Content-Length: 51

{
  "error": "invalid authentication credentials"
}
```

Before we continue, let's quickly have a look at the **tokens** table in our PostgreSQL database to check that the authentication token has been created.

```
$ psql $GREENLIGHT_DB_DSN
Password for user greenlight:
psql (13.1 (Ubuntu 13.1-1.pgdg20.04+1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression: off)
Type "help" for help.

greenlight=> SELECT * FROM tokens WHERE scope = 'authentication';
 \x4390d2ff4af7346dd4238ffccb8a5b18e8c3af9aa8cf57852895ad0f8ee2c50d |      1 | 2021-04-17 11:03:37+02 | authentication
```

That's looking good. We can see that the token is associated with the user with ID **1** (which if you've been following along will be the user **alice@example.com**) and has the correct scope and expiry time.

Additional Information

The Authorization header

Occasionally you might come across other APIs or tutorials where authentication tokens are sent back to the client in an **Authorization** header, rather than in the response body like we are in this chapter.

You *can* do that, and in most cases it will probably work fine. But it's important to be conscious that you making a *willful violation* of the HTTP specifications: **Authorization** is a *request* header, not a response header.

Authenticating Requests

Now that our clients have a way to exchange their credentials for an authentication token, let's look at how we can use that token to *authenticate them*, so we know exactly which user a request is coming from.

Essentially, once a client has an authentication token we will expect them to include it with all subsequent requests in an **Authorization** header, like so:

```
Authorization: Bearer IEYZQUBEMPPAKPOAWTPV6YJ6RM
```

When we receive these requests, we'll use a new **authenticate()** middleware method to execute the following logic:

- If the authentication token is not valid, then we will send the client a **401 Unauthorized** response and an error message to let them know that their token is malformed or invalid.
- If the authentication token is valid, we will look up the user details and add their details to the *request context*.
- If no **Authorization** header was provided at all, then we will add the details for an *anonymous user* to the request context instead.

Creating the anonymous user

Let's start with the final bullet point, and first define an *anonymous user* in our **internal/data/users.go** file, like so:

File: internal/data/users.go

```
package data

...

// Declare a new AnonymousUser variable.
var AnonymousUser = &User{}

type User struct {
    ID          int64      `json:"id"`
    CreatedAt   time.Time  `json:"created_at"`
    Name        string     `json:"name"`
    Email       string     `json:"email"`
    Password    password   `json:"-"`
    Activated   bool       `json:"activated"`
    Version     int        `json:"-"`
}

// Check if a User instance is the AnonymousUser.
func (u *User) IsAnonymous() bool {
    return u == AnonymousUser
}

...
```

So here we've created a new **AnonymousUser** variable, which holds a pointer to a **User** struct representing an *inactivated user with no ID, name, email or password*.

We've also implemented an **IsAnonymous()** method on the **User** struct, so whenever we have a **User** instance we can easily check whether it is the **AnonymousUser** instance or not. For example:

```
data.AnonymousUser.IsAnonymous() // → Returns true

otherUser := &data.User{}
otherUser.IsAnonymous()           // → Returns false
```

Reading and writing to the request context

The other setup step, before we get into creating the **authenticate()** middleware itself, relates to *storing the user details in the request context*.

We discussed what request context is and how to use it in detail in *Let's Go*, and if any of this feels unfamiliar then I recommend re-reading that section of the book before continuing. But as a quick reminder:

- Every **http.Request** that our application processes has a **context.Context** embedded in it, which we can use to store key/value pairs containing arbitrary data during the lifetime

of the request. In this case we want to store a `User` struct containing the current user's information.

- Any values stored in the request context have the type `interface{}`. This means that after retrieving a value from the request context you need to assert it back to its original type before using it.
- It's good practice to use your own custom type for the request context keys. This helps prevent naming collisions between your code and any third-party packages which are also using the request context to store information.

To help with this, let's create a new `cmd/api/context.go` file containing some helper methods for reading/writing the `User` struct to and from the request context.

If you're following along, go ahead and create the new file:

```
$ touch cmd/api/context.go
```

And then add the following code:

File: cmd/api/context.go

```
package main

import (
    "context"
    "net/http"

    "greenlight.alexedwards.net/internal/data"
)

// Define a custom contextKey type, with the underlying type string.
type contextKey string

// Convert the string "user" to a contextKey type and assign it to the userContextKey
// constant. We'll use this constant as the key for getting and setting user information
// in the request context.
const userContextKey = contextKey("user")

// The contextSetUser() method returns a new copy of the request with the provided
// User struct added to the context. Note that we use our userContextKey constant as the
// key.
func (app *application) contextSetUser(r *http.Request, user *data.User) *http.Request {
    ctx := context.WithValue(r.Context(), userContextKey, user)
    return r.WithContext(ctx)
}

// The contextGetUser() retrieves the User struct from the request context. The only
// time that we'll use this helper is when we logically expect there to be User struct
// value in the context, and if it doesn't exist it will firmly be an 'unexpected' error.
// As we discussed earlier in the book, it's OK to panic in those circumstances.
func (app *application) contextGetUser(r *http.Request) *data.User {
    user, ok := r.Context().Value(userContextKey).(*data.User)
    if !ok {
        panic("missing user value in request context")
    }

    return user
}
```

Creating the authentication middleware

Now that we've got those things in place, we're ready to start work on our `authenticate()` middleware itself.

Open up your `cmd/api/middleware.go` file, and add the following code:

File: cmd/api/middleware.go

```
package main

import (
    "errors" // New import
    "fmt"
    "net"
    "net/http"
    "strings" // New import
    "sync"
```

```

    "time"

    "greenlight.alexedwards.net/internal/data" // New import
    "greenlight.alexedwards.net/internal/validator" // New import

    "golang.org/x/time/rate"
)

...

func (app *application) authenticate(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        // Add the "Vary: Authorization" header to the response. This indicates to any
        // caches that the response may vary based on the value of the Authorization
        // header in the request.
        w.Header().Add("Vary", "Authorization")

        // Retrieve the value of the Authorization header from the request. This will
        // return the empty string "" if there is no such header found.
        authorizationHeader := r.Header.Get("Authorization")

        // If there is no Authorization header found, use the contextSetUser() helper
        // that we just made to add the AnonymousUser to the request context. Then we
        // call the next handler in the chain and return without executing any of the
        // code below.
        if authorizationHeader == "" {
            r = app.contextSetUser(r, data.AnonymousUser)
            next.ServeHTTP(w, r)
            return
        }

        // Otherwise, we expect the value of the Authorization header to be in the format
        // "Bearer <token>". We try to split this into its constituent parts, and if the
        // header isn't in the expected format we return a 401 Unauthorized response
        // using the invalidAuthenticationTokenResponse() helper (which we will create
        // in a moment).
        headerParts := strings.Split(authorizationHeader, " ")
        if len(headerParts) != 2 || headerParts[0] != "Bearer" {
            app.invalidAuthenticationTokenResponse(w, r)
            return
        }

        // Extract the actual authentication token from the header parts.
        token := headerParts[1]

        // Validate the token to make sure it is in a sensible format.
        v := validator.New()

        // If the token isn't valid, use the invalidAuthenticationTokenResponse()
        // helper to send a response, rather than the failedValidationResponse() helper
        // that we'd normally use.
        if data.ValidateTokenPlaintext(v, token); !v.Valid() {
            app.invalidAuthenticationTokenResponse(w, r)
            return
        }

        // Retrieve the details of the user associated with the authentication token,
        // again calling the invalidAuthenticationTokenResponse() helper if no
        // matching record was found. IMPORTANT: Notice that we are using
        // ScopeAuthentication as the first parameter here.
        user, err := app.models.Users.GetForToken(data.ScopeAuthentication, token)
        if err != nil {
            switch {
            case errors.Is(err, data.ErrRecordNotFound):
                app.invalidAuthenticationTokenResponse(w, r)
            default:

```

```

        app.serverErrorResponse(w, r, err)
    }
    return
}

// Call the contextSetUser() helper to add the user information to the request
// context.
r = app.contextSetUser(r, user)

// Call the next handler in the chain.
next.ServeHTTP(w, r)
})
}

```

There's quite a lot of code there, so to help clarify things, let's quickly reiterate the actions that this middleware is taking:

- If a valid authentication token is provided in the **Authorization** header, then a **User** struct containing the corresponding user details will be stored in the request context.
- If no **Authorization** header is provided at all, our **AnonymousUser** struct will be stored in the request context.
- If the **Authorization** header is provided, but it's malformed or contains an invalid value, the client will be sent a **401 Unauthorized** response using the **invalidAuthenticationTokenResponse()** helper.

Talking of which, let's head to our **cmd/api/errors.go** file and create that helper as follows:

```

File: cmd/api/errors.go

package main

...

func (app *application) invalidAuthenticationTokenResponse(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("WWW-Authenticate", "Bearer")

    message := "invalid or missing authentication token"
    app.errorResponse(w, r, http.StatusUnauthorized, message)
}

```

Note: We're including a **WWW-Authenticate: Bearer** header here to help inform or remind the client that we expect them to authenticate using a bearer token.

Finally, we need to add the **authenticate()** middleware to our handler chain. We want to use this middleware on *all* requests — after our panic recovery and rate limiter middleware, but before our router.

Go ahead and update the `cmd/api/routes.go` file accordingly:

File: `cmd/api/routes.go`

```
package main

...

func (app *application) routes() http.Handler {
    router := httprouter.New()

    router.NotFound = http.HandlerFunc(app.notFoundResponse)
    router.MethodNotAllowed = http.HandlerFunc(app.methodNotAllowedResponse)

    router.HandleFunc(http.MethodGet, "/v1/healthcheck", app.healthcheckHandler)

    router.HandleFunc(http.MethodGet, "/v1/movies", app.listMoviesHandler)
    router.HandleFunc(http.MethodPost, "/v1/movies", app.createMovieHandler)
    router.HandleFunc(http.MethodGet, "/v1/movies/:id", app.showMovieHandler)
    router.HandleFunc(http.MethodPatch, "/v1/movies/:id", app.updateMovieHandler)
    router.HandleFunc(http.MethodDelete, "/v1/movies/:id", app.deleteMovieHandler)

    router.HandleFunc(http.MethodPost, "/v1/users", app.registerUserHandler)
    router.HandleFunc(http.MethodPut, "/v1/users/activated", app.activateUserHandler)

    router.HandleFunc(http.MethodPost, "/v1/tokens/authentication", app.createAuthenticationTokenHandler)

    // Use the authenticate() middleware on all requests.
    return app.recoverPanic(app.rateLimit(app.authenticate(router)))
}
```

Demonstration

Let's test this out by first making a request with *no* `Authorization` header. Behind the scenes, our `authenticate()` middleware will add the `AnonymousUser` to the request context and the request should complete successfully. Like so:

```
$ curl localhost:4000/v1/healthcheck
{
  "status": "available",
  "system_info": {
    "environment": "development",
    "version": "1.0.0"
  }
}
```

Then let's try the same thing but with a valid authentication token in the `Authorization` header. This time, the relevant user details should be added to the request context, and we should again get a successful response. For example:


```
$ curl -d '{"email": "alice@example.com", "password": "pa55word"}' localhost:4000/v1/tokens/authentication
{
  "authentication_token": {
    "token": "FXCZM44TVLC6ML2NXTOW50HFUE",
    "expiry": "2021-04-17T12:20:30.02833444+02:00"
  }
}

$ curl -H "Authorization: Bearer FXCZM44TVLC6ML2NXTOW50HFUE" localhost:4000/v1/healthcheck
{
  "status": "available",
  "system_info": {
    "environment": "development",
    "version": "1.0.0"
  }
}
```

Hint: If you get an error response here, make sure that you're using the correct authentication token from the first request in the second request.

In contrast, you can also try sending some requests with an invalid authentication token or a malformed **Authorization** header. In these cases you should get a **401 Unauthorized** response, like so:

```
$ curl -i -H "Authorization: Bearer XXXXXXXXXXXXXXXXXXXXXXXX" localhost:4000/v1/healthcheck
HTTP/1.1 401 Unauthorized
Content-Type: application/json
Vary: Authorization
Www-Authenticate: Bearer
Date: Fri, 16 Apr 2021 10:23:06 GMT
Content-Length: 56

{
  "error": "invalid or missing authentication token"
}

$ curl -i -H "Authorization: INVALID" localhost:4000/v1/healthcheck
HTTP/1.1 401 Unauthorized
Content-Type: application/json
Vary: Authorization
Www-Authenticate: Bearer
Date: Fri, 16 Apr 2021 10:23:26 GMT
Content-Length: 56

{
  "error": "invalid or missing authentication token"
}
```

Permission-based Authorization

By the time a request leaves our `authenticate()` middleware, there are now two possible states for the request context. Either:

- The request context contains a `User` struct (representing a valid, authenticated, user).
- Or the request context contains an `AnonymousUser` struct.

In this section of the book, we're going to take this to the next natural stage and look at how to perform different *authorization* checks to restrict access to our API endpoints. Specifically, you'll learn how to:

- Add checks so that only *activated users* are able to access the various `/v1/movies**` endpoints.
- Implement a *permission-based authorization* pattern, which provides fine-grained control over exactly *which users can access which endpoints*.

Requiring User Activation

As we mentioned a moment ago, the first thing we're going to do in terms of authorization is restrict access to our `/v1/movies**` endpoints — so that they can only be accessed by users who are authenticated (not anonymous), and who have activated their account.

Carrying out these kinds of checks is an ideal task for some middleware, so let's jump in and make a new `requireActivatedUser()` middleware method to handle this. In this middleware, we want to extract the `User` struct from the request context and then check the `IsAnonymous()` method and `Activated` field to determine whether the request should continue or not.

Specifically:

- If the user is anonymous we should send a `401 Unauthorized` response and an error message saying “you must be authenticated to access this resource”.
- If the user is not anonymous (i.e. they have authenticated successfully and we know who they are), but they are *not activated* we should send a `403 Forbidden` response and an error message saying “your user account must be activated to access this resource”.

Remember: A `401 Unauthorized` response should be used when you have missing or bad authentication, and a `403 Forbidden` response should be used afterwards, when the user is authenticated but isn't allowed to perform the requested operation.

So first, let's head to our `cmd/api/errors.go` file and add a couple of new helpers for sending those error messages. Like so:

File: cmd/api/errors.go

```
package main

...

func (app *application) authenticationRequiredResponse(w http.ResponseWriter, r *http.Request) {
    message := "you must be authenticated to access this resource"
    app.errorResponse(w, r, http.StatusUnauthorized, message)
}

func (app *application) inactiveAccountResponse(w http.ResponseWriter, r *http.Request) {
    message := "your user account must be activated to access this resource"
    app.errorResponse(w, r, http.StatusForbidden, message)
}
```

And then let's create the new `requireActivatedUser()` middleware for carrying out the checks. The code we need is nice and succinct:

File: cmd/api/middleware.go

```
package main

...

func (app *application) requireActivatedUser(next http.HandlerFunc) http.HandlerFunc {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        // Use the contextGetUser() helper that we made earlier to retrieve the user
        // information from the request context.
        user := app.contextGetUser(r)

        // If the user is anonymous, then call the authenticationRequiredResponse() to
        // inform the client that they should authenticate before trying again.
        if user.IsAnonymous() {
            app.authenticationRequiredResponse(w, r)
            return
        }

        // If the user is not activated, use the inactiveAccountResponse() helper to
        // inform them that they need to activate their account.
        if !user.Activated {
            app.inactiveAccountResponse(w, r)
            return
        }

        // Call the next handler in the chain.
        next.ServeHTTP(w, r)
    })
}
```

Notice here that our `requireActivatedUser()` middleware has a slightly different signature to the other middleware we've built in this book. Instead of accepting and returning a `http.Handler`, it accepts and returns a `http.HandlerFunc`.

This is a small change, but it makes it possible to wrap our `/v1/movie**` handler functions directly with this middleware, without needing to make any further conversions.

Go ahead and update the `cmd/api/routes.go` file to do exactly that, as follows:

File: `cmd/api/routes.go`

```
package main

...

func (app *application) routes() http.Handler {
    router := httprouter.New()

    router.NotFound = http.HandlerFunc(app.notFoundResponse)
    router.MethodNotAllowed = http.HandlerFunc(app.methodNotAllowedResponse)

    router.HandleFunc(http.MethodGet, "/v1/healthcheck", app.healthcheckHandler)

    // Use the requireActivatedUser() middleware on our five /v1/movies** endpoints.
    router.HandleFunc(http.MethodGet, "/v1/movies", app.requireActivatedUser(app.listMoviesHandler))
    router.HandleFunc(http.MethodPost, "/v1/movies", app.requireActivatedUser(app.createMovieHandler))
    router.HandleFunc(http.MethodGet, "/v1/movies/:id", app.requireActivatedUser(app.showMovieHandler))
    router.HandleFunc(http.MethodPatch, "/v1/movies/:id", app.requireActivatedUser(app.updateMovieHandler))
    router.HandleFunc(http.MethodDelete, "/v1/movies/:id", app.requireActivatedUser(app.deleteMovieHandler))

    router.HandleFunc(http.MethodPost, "/v1/users", app.registerUserHandler)
    router.HandleFunc(http.MethodPut, "/v1/users/activated", app.activateUserHandler)

    router.HandleFunc(http.MethodPost, "/v1/tokens/authentication", app.createAuthenticationTokenHandler)

    return app.recoverPanic(app.rateLimit(app.authenticate(router)))
}
```

Demonstration

Alright, let's try out these changes!

We'll begin by calling the `GET /v1/movies/:id` endpoint as an anonymous user. When doing this you should now receive a `401 Unauthorized` response, like so:

```
$ curl -i localhost:4000/v1/movies/1
HTTP/1.1 401 Unauthorized
Content-Type: application/json
Vary: Authorization
Www-Authenticate: Bearer
Date: Fri, 16 Apr 2021 15:59:33 GMT
Content-Length: 66

{
  "error": "you must be authenticated to access this resource"
}
```

Next, let's try making a request as a user who has an account, but has not yet activated. If you've been following along, you should be able to use the `alice@example.com` user to do

this, like so:

```
$ BODY='{"email": "alice@example.com", "password": "pa55word"}'
$ curl -d "$BODY" localhost:4000/v1/tokens/authentication
{
  "authentication_token": {
    "token": "204YHHWDHVWWDNKN2UZR722BU",
    "expiry": "2021-04-17T18:03:09.598843181+02:00"
  }
}

$ curl -i -H "Authorization: Bearer 204YHHWDHVWWDNKN2UZR722BU" localhost:4000/v1/movies/1
HTTP/1.1 403 Forbidden
Content-Type: application/json
Vary: Authorization
Date: Fri, 16 Apr 2021 16:03:45 GMT
Content-Length: 76

{
  "error": "your user account must be activated to access this resource"
}
```

Great, we can see that this now results in a **403 Forbidden** response from our new `inactiveAccountResponse()` helper.

Finally, let's try making a request as an activated user.

If you're coding-along, you might like to quickly connect to your PostgreSQL database and double-check which users are already activated.

```
$ psql $GREENLIGHT_DB_DSN
psql (13.2 (Ubuntu 13.2-1.pgdg20.04+1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression: off)
Type "help" for help.

greenlight=> SELECT email FROM users WHERE activated = true;
 email
-----
 faith@example.com
(1 row)
```

In my case the only activated user is `faith@example.com`, so let's try making a request as them. When making a request as an activated user, all the checks in our `requireActivatedUser()` middleware will pass and you should get a successful response. Similar to this:

```
$ BODY='{ "email": "faith@example.com", "password": "pa55word" }'
$ curl -d "$BODY" localhost:4000/v1/tokens/authentication
{
  "authentication_token": {
    "token": "ZFIKQ344EYM5KEP6JL2RHLPJQ",
    "expiry": "2021-04-17T18:04:57.513348573+02:00"
  }
}

$ curl -H "Authorization: Bearer ZFIKQ344EYM5KEP6JL2RHLPJQ" localhost:4000/v1/movies/1
{
  "movie": {
    "id": 1,
    "title": "Moana",
    "year": 2016,
    "runtime": "107 mins",
    "genres": [
      "animation",
      "adventure"
    ],
    "version": 1
  }
}
```

Splitting up the middleware

At the moment we have one piece of middleware doing two checks: first it checks that the user is authenticated (not anonymous), and second it checks that they are activated.

But it's possible to imagine a scenario where you *only* want to check that a user is authenticated, and you don't care whether they are activated or not. To assist with this, you might want to introduce an additional `requireAuthenticatedUser()` middleware as well as the current `requireActivatedUser()` middleware.

However, there would be some overlap between these two middlewares, as they would both be checking whether a user is authenticated or not. A neat way to avoid this duplication is to have your `requireActivatedUser()` middleware automatically call the `requireAuthenticatedUser()` middleware.

It's hard to describe in words how this pattern works, so I'll demonstrate. If you're following along go ahead and update your `cmd/api/middleware.go` file like so:

File: cmd/api/middleware.go

```
package main

...

// Create a new requireAuthenticatedUser() middleware to check that a user is not
// anonymous.
func (app *application) requireAuthenticatedUser(next http.HandlerFunc) http.HandlerFunc {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        user := app.context.GetUser(r)

        if user.IsAnonymous() {
            app.authenticationRequiredResponse(w, r)
            return
        }

        next.ServeHTTP(w, r)
    })
}

// Checks that a user is both authenticated and activated.
func (app *application) requireActivatedUser(next http.HandlerFunc) http.HandlerFunc {
    // Rather than returning this http.HandlerFunc we assign it to the variable fn.
    fn := http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        user := app.context.GetUser(r)

        // Check that a user is activated.
        if !user.Activated {
            app.inactiveAccountResponse(w, r)
            return
        }

        next.ServeHTTP(w, r)
    })

    // Wrap fn with the requireAuthenticatedUser() middleware before returning it.
    return app.requireAuthenticatedUser(fn)
}
```

The way that we've set this up, our `requireActivatedUser()` middleware now automatically calls the `requireAuthenticatedUser()` middleware *before being executed itself*. In our application this makes a lot of sense — we shouldn't be checking if a user is activated unless we know exactly who they are!

You can go ahead and run the application again now — everything should compile and continue to work just like it did before.

Additional Information

In-handler checks

If you only have a couple of endpoints where you want to perform authorization checks, then rather than using middleware it can often be easier to do the checks inside the relevant handlers instead. For example:

```
func (app *application) exampleHandler(w http.ResponseWriter, r *http.Request) {  
    user := app.contextGetUser(r)  
  
    if user.IsAnonymous() {  
        app.authenticationRequiredResponse(w, r)  
        return  
    }  
  
    if !user.Activated {  
        app.inactiveAccountResponse(w, r)  
        return  
    }  
  
    // The rest of the handler logic goes here...  
}
```

Setting up the Permissions Database Table

Restricting our API so that movie data can only be accessed and edited by activated users is useful, but sometimes you might need a more granular level of control. For example, in our case we might be happy for ‘regular’ users of our API to *read* the movie data (so long as they are activated), but we want to restrict *write* access to a smaller subset of trusted users.

In this chapter we’re going to introduce the concept of *permissions* to our application, so that only users who have a specific permission can perform specific operations. In our case, we’re going to create two permissions: a **movies:read** permission which will allow a user to fetch and filter movies, and a **movies:write** permission which will allow users to create, edit and delete movies.

The required permissions will align with our API endpoints like so:

Method	URL Pattern	Required permission
GET	/v1/healthcheck	–
GET	/v1/movies	movies:read
POST	/v1/movies	movies:write
GET	/v1/movies/:id	movies:read
PATCH	/v1/movies/:id	movies:write
DELETE	/v1/movies/:id	movies:write
POST	/v1/users	–
PUT	/v1/users/activated	–
POST	/v1/tokens/authentication	–

Relationship between permissions and users

The relationship between permissions and users is a great example of a *many-to-many* relationship. One user may have *many permissions*, and the same permission may belong to

many users.

The classic way to manage a many-to-many relationship in a relational database like PostgreSQL is to create a *joining table* between the two entities. I'll quickly explain how this works.

Let's say that we are storing our user data in a `users` table which looks like this:

id	email	...
1	alice@example.com	...
2	bob@example.com	...

And our permissions data is stored in a `permissions` table like this:

id	code
1	movies:read
2	movies:write

Then we can create a joining table called `users_permissions` to store the information about *which users have which permissions*, similar to this:

user_id	permission_id
1	1
2	1
2	2

In the example above, the user `alice@example.com` (user ID `1`) has the `movies:read` (permission ID `1`) permission only, whereas `bob@example.com` (user ID `2`) has both the `movies:read` and `movies:write` permissions.

Just like the one-to-many relationship that we looked at earlier in the book, you may want to query this relationship from both sides in your database models. For example, in your database models you might want to create the following methods:

```
PermissionModel.GetAllForUser(user)      → Retrieve all permissions for a user
UserModel.GetAllForPermission(permission) → Retrieve all users with a specific permission
```

Creating the SQL migrations

Let's put this into practice and make a SQL migration which creates new **permissions** and **users_permissions** tables in our database, following the pattern that we've just described above.

Go ahead and run the following command to create the migration files:

```
$ migrate create -seq -ext .sql -dir ./migrations add_permissions
/home/alex/Projects/greenlight/migrations/000006_add_permissions.up.sql
/home/alex/Projects/greenlight/migrations/000006_add_permissions.down.sql
```

And then add the following SQL statements to the 'up' migration file:

```
File: migrations/000006_add_permissions.up.sql

CREATE TABLE IF NOT EXISTS permissions (
  id bigserial PRIMARY KEY,
  code text NOT NULL
);

CREATE TABLE IF NOT EXISTS users_permissions (
  user_id bigint NOT NULL REFERENCES users ON DELETE CASCADE,
  permission_id bigint NOT NULL REFERENCES permissions ON DELETE CASCADE,
  PRIMARY KEY (user_id, permission_id)
);

-- Add the two permissions to the table.
INSERT INTO permissions (code)
VALUES
  ('movies:read'),
  ('movies:write');
```

There are a couple of important things to point out here:

- The **PRIMARY KEY (user_id, permission_id)** line sets a *composite primary key* on our **users_permissions** table, where the primary key is made up of both the **users_id** and **permission_id** columns. Setting this as the primary key essentially means that the same user/permission combination can only appear once in the table and cannot be duplicated.
- When creating the **users_permissions** table we use the **REFERENCES user** syntax to create a foreign key constraint against the primary key of our **users** table, which ensures

that any value in the `user_id` column has a corresponding entry in our `users` table. And likewise, we use the `REFERENCES permissions` syntax to ensure that the `permission_id` column has a corresponding entry in the `permissions` table.

Let's also add the necessary `DROP TABLE` statements to the 'down' migration file, like so:

```
File: migrations/000006_add_permissions.down.sql
```

```
DROP TABLE IF EXISTS users_permissions;  
DROP TABLE IF EXISTS permissions;
```

Now that's done, please go ahead and run the migration:

```
$ migrate -path ./migrations -database $GREENLIGHT_DB_DSN up  
6/u add_permissions (22.74009ms)
```

Setting up the Permissions Model

Next let's head to our `internal/data` package and add a `PermissionModel` to manage the interactions with our new tables. For now, the only thing we want to include in this model is a `GetAllForUser()` method to *return all permission codes for a specific user*. The idea is that we'll be able to use this in our handlers and middleware like so:

```
// Return a slice of the permission codes for the user with ID = 1. This would return
// something like []string{"movies:read", "movies:write"}.
app.models.Permissions.GetAllForUser(1)
```

Behind the scenes, the SQL statement that we need to fetch the permission codes for a specific user looks like this:

```
SELECT permissions.code
FROM permissions
INNER JOIN users_permissions ON users_permissions.permission_id = permissions.id
INNER JOIN users ON users_permissions.user_id = users.id
WHERE users.id = $1
```

In this query we are using the `INNER JOIN` clause to join our `permissions` table to our `users_permissions` table, and then using it again to join *that* to the `users` table. Then we use the `WHERE` clause to filter the result, leaving only rows which relate to a specific user ID.

Let's go ahead and setup the new `PermissionModel`. First create a new `internal/data/permissions.go` file:

```
$ touch internal/data/permissions.go
```

And then add the following code:

```
File: internal/data/permissions.go

package data

import (
    "context"
    "database/sql"
    "time"
)

// Define a Permissions slice, which we will use to hold the permission codes (like
// "movies:read" and "movies:write") for a single user
```

```

// movies:read and movies:write ) for a single user.
type Permissions []string

// Add a helper method to check whether the Permissions slice contains a specific
// permission code.
func (p Permissions) Include(code string) bool {
    for i := range p {
        if code == p[i] {
            return true
        }
    }
    return false
}

// Define the PermissionModel type.
type PermissionModel struct {
    DB *sql.DB
}

// The GetAllForUser() method returns all permission codes for a specific user in a
// Permissions slice. The code in this method should feel very familiar --- it uses the
// standard pattern that we've already seen before for retrieving multiple data rows in
// an SQL query.
func (m PermissionModel) GetAllForUser(userID int64) (Permissions, error) {
    query := `
        SELECT permissions.code
        FROM permissions
        INNER JOIN users_permissions ON users_permissions.permission_id = permissions.id
        INNER JOIN users ON users_permissions.user_id = users.id
        WHERE users.id = $1`

    ctx, cancel := context.WithTimeout(context.Background(), 3*time.Second)
    defer cancel()

    rows, err := m.DB.QueryContext(ctx, query, userID)
    if err != nil {
        return nil, err
    }
    defer rows.Close()

    var permissions Permissions

    for rows.Next() {
        var permission string

        err := rows.Scan(&permission)
        if err != nil {
            return nil, err
        }

        permissions = append(permissions, permission)
    }
    if err = rows.Err(); err != nil {
        return nil, err
    }

    return permissions, nil
}

```

Then the last thing we need to do is add the `PermissionModel` to our parent `Model` struct, so it's available to our handlers and middleware. Like so:

File: internal/data/models.go

```
package data

...

type Models struct {
    Movies      MovieModel
    Permissions PermissionModel // Add a new Permissions field.
    Tokens      TokenModel
    Users       UserModel
}

func NewModels(db *sql.DB) Models {
    return Models{
        Movies:      MovieModel{DB: db},
        Permissions: PermissionModel{DB: db}, // Initialize a new PermissionModel instance.
        Tokens:      TokenModel{DB: db},
        Users:       UserModel{DB: db},
    }
}
```


Checking Permissions

Now that our `PermissionModel` is set up, let's look at how we can use it to restrict access to our API endpoints.

Conceptually, what we need to do here isn't too complicated.

- We'll make a new `requirePermission()` middleware which accepts a specific permission code like `"movies:read"` as an argument.
- In this middleware we'll retrieve the current user from the request context, and call the `app.models.Permissions.GetAllForUser()` method (which we just made) to get a slice of their permissions.
- Then we can check to see if the slice contains the specific permission code needed. If it doesn't, we should send the client a `403 Forbidden` response.

To put this into practice, let's first make a new `notPermittedResponse()` helper function for sending the `403 Forbidden` response. Like so:

File: `cmd/api/errors.go`

```
package main

...

func (app *application) notPermittedResponse(w http.ResponseWriter, r *http.Request) {
    message := "your user account doesn't have the necessary permissions to access this resource"
    app.errorResponse(w, r, http.StatusForbidden, message)
}
```

Then let's head to our `cmd/api/middleware.go` file and create the new `requirePermission()` middleware method.

We're going to set this up so that the `requirePermission()` middleware automatically wraps our existing `requireActivatedUser()` middleware, which in turn — don't forget — wraps our `requireAuthenticatedUser()` middleware.

This is important — it means that when we use the `requirePermission()` middleware we'll actually be carrying out *three checks* which together ensure that the request is from an *authenticated (non-anonymous), activated user, who has a specific permission*.

Let's go ahead and create this in the `cmd/api/middleware.go` file like so:

File: cmd/api/middleware.go

```
package main

...

// Note that the first parameter for the middleware function is the permission code that
// we require the user to have.
func (app *application) requirePermission(code string, next http.HandlerFunc) http.HandlerFunc {
    fn := func(w http.ResponseWriter, r *http.Request) {
        // Retrieve the user from the request context.
        user := app.contextGetUser(r)

        // Get the slice of permissions for the user.
        permissions, err := app.models.Permissions.GetAllForUser(user.ID)
        if err != nil {
            app.serverErrorResponse(w, r, err)
            return
        }

        // Check if the slice includes the required permission. If it doesn't, then
        // return a 403 Forbidden response.
        if !permissions.Include(code) {
            app.notPermittedResponse(w, r)
            return
        }

        // Otherwise they have the required permission so we call the next handler in
        // the chain.
        next.ServeHTTP(w, r)
    }

    // Wrap this with the requireActivatedUser() middleware before returning it.
    return app.requireActivatedUser(fn)
}
```

Once that's done, the final step is to update our `cmd/api/routes.go` file to utilize the new middleware on the necessary endpoints.

Go ahead and update the routes so that our API requires the `"movies:read"` permission for the endpoints that fetch movie data, and the `"movies:write"` permission for the endpoints that create, edit or delete a movie.

File: cmd/api/routes.go

```
package main

...

func (app *application) routes() http.Handler {
    router := httprouter.New()

    router.NotFound = http.HandlerFunc(app.notFoundResponse)
    router.MethodNotAllowed = http.HandlerFunc(app.methodNotAllowedResponse)

    router.HandleFunc(http.MethodGet, "/v1/healthcheck", app.healthcheckHandler)

    // Use the requirePermission() middleware on each of the /v1/movies** endpoints,
    // passing in the required permission code as the first parameter.
    router.HandleFunc(http.MethodGet, "/v1/movies", app.requirePermission("movies:read", app.listMoviesHandler))
    router.HandleFunc(http.MethodPost, "/v1/movies", app.requirePermission("movies:write", app.createMovieHandler))
    router.HandleFunc(http.MethodGet, "/v1/movies/:id", app.requirePermission("movies:read", app.showMovieHandler))
    router.HandleFunc(http.MethodPatch, "/v1/movies/:id", app.requirePermission("movies:write", app.updateMovieHandler))
    router.HandleFunc(http.MethodDelete, "/v1/movies/:id", app.requirePermission("movies:write", app.deleteMovieHandler))

    router.HandleFunc(http.MethodPost, "/v1/users", app.registerUserHandler)
    router.HandleFunc(http.MethodPut, "/v1/users/activated", app.activateUserHandler)

    router.HandleFunc(http.MethodPost, "/v1/tokens/authentication", app.createAuthenticationTokenHandler)

    return app.recoverPanic(app.rateLimit(app.authenticate(router)))
}
```

Demonstration

Showing this in action is a bit awkward because, if you've been following along, none of the users in our database currently have any permissions set for them.

To help demonstrate this new functionality, let's open **psql** and add some permissions. Specifically, we will:

- Activate the user **alice@example.com**.
- Give *all* users the **"movies:read"** permission.
- Give the user **faith@example.com** the **"movies:write"** permission.

If you're following along, open your **psql** prompt:

```
$ psql $GREENLIGHT_DB_DSN
Password for user greenlight:
psql (13.1 (Ubuntu 13.1-1.pgdg20.04+1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression: off)
Type "help" for help.

greenlight=>
```

And execute the following statements:

```
-- Set the activated field for alice@example.com to true.
UPDATE users SET activated = true WHERE email = 'alice@example.com';

-- Give all users the 'movies:read' permission
INSERT INTO users_permissions
SELECT id, (SELECT id FROM permissions WHERE code = 'movies:read') FROM users;

-- Give faith@example.com the 'movies:write' permission
INSERT INTO users_permissions
VALUES (
  (SELECT id FROM users WHERE email = 'faith@example.com'),
  (SELECT id FROM permissions WHERE code = 'movies:write')
);

-- List all activated users and their permissions.
SELECT email, array_agg(permissions.code) as permissions
FROM permissions
INNER JOIN users_permissions ON users_permissions.permission_id = permissions.id
INNER JOIN users ON users_permissions.user_id = users.id
WHERE users.activated = true
GROUP BY email;
```

Once that completes, you should see a list of the currently activated users and their permissions, similar to this:

```
email | permissions
-----+-----
alice@example.com | {movies:read}
faith@example.com | {movies:read,movies:write}
(2 rows)
```

Note: In that final SQL query, we're using the aggregation function `array_agg()` and a `GROUP BY` clause to output the permissions associated with each email address as an array.

Now that our users have some permissions assigned to them, we're ready to try this out.

To begin with, let's try making some requests as `alice@example.com` to our `GET /v1/movies/1` and `DELETE /v1/movies/1` endpoints. The first request should work correctly, but the second should fail because the user doesn't have the necessary `movies:write` permission.

```
$ BODY='{"email": "alice@example.com", "password": "pa55word"}'
$ curl -d "$BODY" localhost:4000/v1/tokens/authentication
{
  "authentication_token": {
    "token": "OPFXEPOYZWMGNWXWKMYIMEGATU",
    "expiry": "2021-04-17T20:49:39.963768416+02:00"
  }
}

$ curl -H "Authorization: Bearer OPFXEPOYZWMGNWXWKMYIMEGATU" localhost:4000/v1/movies/1
{
  "movie": {
    "id": 1,
    "title": "Moana",
    "year": 2016,
    "runtime": "107 mins",
    "genres": [
      "animation",
      "adventure"
    ],
    "version": 1
  }
}

$ curl -X DELETE -H "Authorization: Bearer OPFXEPOYZWMGNWXWKMYIMEGATU" localhost:4000/v1/movies/1
{
  "error": "your user account doesn't have the necessary permissions to access this resource"
}
```

Great, that's working just as we expected — the **DELETE** operation is blocked because **alice@example.com** doesn't have the necessary **movies:write** permission.

In contrast, let's try the same operation but with **faith@example.com** as the user. This time the **DELETE** operation should work correctly, like so:

```
$ BODY='{"email": "faith@example.com", "password": "pa55word"}'
$ curl -d "$BODY" localhost:4000/v1/tokens/authentication
{
  "authentication_token": {
    "token": "E42XD50BBB04MPUPYGLLY2GURE",
    "expiry": "2021-04-17T20:51:14.924813208+02:00"
  }
}

$ curl -X DELETE -H "Authorization: Bearer E42XD50BBB04MPUPYGLLY2GURE" localhost:4000/v1/movies/1
{
  "message": "movie successfully deleted"
}
```

Granting Permissions

Our permissions model and authorization middleware are now functioning well. But — at the moment — when a new user registers an account they *don't have any permissions*. In this chapter we're going to change that so that new users are automatically *granted* the `"movies:read"` permission by default.

Updating the permissions model

In order to grant permissions to a user we'll need to update our `PermissionModel` to include an `AddForUser()` method, which adds *one or more permission codes for a specific user* to our database. The idea is that we will be able to use it in our handlers like this:

```
// Add the "movies:read" and "movies:write" permissions for the user with ID = 2.
app.models.Permissions.AddForUser(2, "movies:read", "movies:write")
```

Behind the scenes, the SQL statement that we need to insert this data looks like this:

```
INSERT INTO users_permissions
SELECT $1, permissions.id FROM permissions WHERE permissions.code = ANY($2)
```

In this query the `$1` parameter will be the user's ID, and the `$2` parameter will be a *PostgreSQL array* of the permission codes that we want to add for the user, like `{'movies:read', 'movies:write'}`.

So what's happening here is that the `SELECT ...` statement on the second line creates an 'interim' table with rows made up of the user ID *and the corresponding IDs for the permission codes in the array*. Then we insert the contents of this interim table into our `user_permissions` table.

Let's go ahead and create the `AddForUser()` method in the `internal/data/permissions.go` file:

File: internal/data/permissions.go

```
package data

import (
    "context"
    "database/sql"
    "time"

    "github.com/lib/pq" // New import
)

...

// Add the provided permission codes for a specific user. Notice that we're using a
// variadic parameter for the codes so that we can assign multiple permissions in a
// single call.
func (m PermissionModel) AddForUser(userID int64, codes ...string) error {
    query := `
        INSERT INTO users_permissions
        SELECT $1, permissions.id FROM permissions WHERE permissions.code = ANY($2)`

    ctx, cancel := context.WithTimeout(context.Background(), 3*time.Second)
    defer cancel()

    _, err := m.DB.ExecContext(ctx, query, userID, pq.Array(codes))
    return err
}
```

Updating the registration handler

Now that's in place, let's update our `registerUserHandler` so that new users are automatically granted the `movies:read` permission when they register. Like so:

File: cmd/api/users.go

```
package main

...

func (app *application) registerUserHandler(w http.ResponseWriter, r *http.Request) {
    var input struct {
        Name      string `json:"name"`
        Email     string `json:"email"`
        Password  string `json:"password"`
    }

    err := app.readJSON(w, r, &input)
    if err != nil {
        app.badRequestResponse(w, r, err)
        return
    }

    user := &data.User{
        Name:      input.Name,
        Email:     input.Email,
        Activated: false,
    }
}
```

```

err = user.Password.Set(input.Password)
if err != nil {
    app.serverErrorResponse(w, r, err)
    return
}

v := validator.New()

if data.ValidateUser(v, user); !v.Valid() {
    app.failedValidationResponse(w, r, v.Errors)
    return
}

err = app.models.Users.Insert(user)
if err != nil {
    switch {
    case errors.Is(err, data.ErrDuplicateEmail):
        v.AddError("email", "a user with this email address already exists")
        app.failedValidationResponse(w, r, v.Errors)
    default:
        app.serverErrorResponse(w, r, err)
    }
    return
}

// Add the "movies:read" permission for the new user.
err = app.models.Permissions.AddForUser(user.ID, "movies:read")
if err != nil {
    app.serverErrorResponse(w, r, err)
    return
}

token, err := app.models.Tokens.New(user.ID, 3*24*time.Hour, data.ScopeActivation)
if err != nil {
    app.serverErrorResponse(w, r, err)
    return
}

app.background(func() {
    data := map[string]interface{}{
        "activationToken": token.Plaintext,
        "userID":          user.ID,
    }

    err = app.mailer.Send(user.Email, "user_welcome.tmpl", data)
    if err != nil {
        app.logger.PrintError(err, nil)
    }
})

err = app.writeJSON(w, http.StatusOK, envelope{"user": user}, nil)
if err != nil {
    app.serverErrorResponse(w, r, err)
}
}

...

```

Let's check that this is working correctly by registering a brand-new user with the email address `grace@example.com`:


```
$ BODY='{"name": "Grace Smith", "email": "grace@example.com", "password": "pa55word"}'
$ curl -d "$BODY" localhost:4000/v1/users
{
  "user": {
    "id": 8,
    "created_at": "2021-04-16T21:32:56+02:00",
    "name": "Grace Smith",
    "email": "grace@example.com",
    "activated": false
  }
}
```

If you open **psql**, you should be able to see that they have the **movies:read** permission by running the following SQL query:

```
SELECT email, code FROM users
INNER JOIN users_permissions ON users.id = users_permissions.user_id
INNER JOIN permissions ON users_permissions.permission_id = permissions.id
WHERE users.email = 'grace@example.com';
```

Like so:

```
$ psql $GREENLIGHT_DB_DSN
Password for user greenlight:
psql (13.1 (Ubuntu 13.1-1.pgdg20.04+1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression: off)
Type "help" for help.

greenlight=> SELECT email, code FROM users
greenlight-> INNER JOIN users_permissions ON users.id = users_permissions.user_id
greenlight-> INNER JOIN permissions ON users_permissions.permission_id = permissions.id
greenlight-> WHERE users.email = 'grace@example.com';
      email      |      code
-----+-----
grace@example.com | movies:read
(1 row)
```

Cross Origin Requests

In this section of the book, we're going to switch to a completely new topic and update our application so that it supports cross-origin requests (CORS) from JavaScript.

You'll learn:

- What cross-origin requests are, and why web browsers prevent them by default.
- The difference between *simple* and *preflight* cross-origin requests.
- How to use **Access-Control** headers to allow or disallow specific cross-origin requests.
- About the security considerations you need to be aware of when CORS configuring settings in your application.

An Overview of CORS

Before we dive into any code, or start talking about cross-origin requests specifically, let's talk a moment to define what we mean by the term *origin*.

Basically, if two URLs have the same *scheme*, *host* and *port* (if specified) they are said to share the same origin. To help illustrate this, let's compare the following URLs:

URL A	URL B	Same origin?	Reason
<code>https://foo.com/a</code>	<code>http://foo.com/a</code>	No	Different scheme (<code>http</code> vs <code>https</code>)
<code>http://foo.com/a</code>	<code>http://www.foo.com/a</code>	No	Different host (<code>foo.com</code> vs <code>www.foo.com</code>)
<code>http://foo.com/a</code>	<code>http://foo.com:443/a</code>	No	Different port (no port vs <code>443</code>)
<code>http://foo.com/a</code>	<code>http://foo.com/b</code>	Yes	Only the path is different
<code>http://foo.com/a</code>	<code>http://foo.com/a?b=c</code>	Yes	Only the query string is different
<code>http://foo.com/a#b</code>	<code>http://foo.com/a#c</code>	Yes	Only the fragment is different

Understanding what origins are is important because all web browsers implement a security mechanism known as the *same-origin policy*. There are some very small differences in how browsers implement this policy, but broadly speaking:

- A webpage on one origin can *embed* certain types of resources from another origin in their HTML — including images, CSS, and JavaScript files. For example, doing this in your webpage is OK:

```

```

- A webpage on one origin can *send* data to a different origin. For example, it's OK for a HTML form in a webpage to submit data to a different origin.
- But a webpage on one origin is *not* allowed to *receive* data from a different origin.

This key thing here is the final bullet-point: the same-origin policy prevents a (potentially malicious) website on another origin from *reading* (possibly confidential) information from

your website.

It's important to emphasize that cross-origin *sending of data* is not prevented by the same-origin policy, despite also being dangerous. In fact, this is why CSRF attacks are possible and why we need to take additional steps to prevent them — like using `SameSite` cookies and CSRF tokens.

As a developer, the time that you're most likely to run up against the same-origin policy is when making cross-origin requests from JavaScript running in a browser.

For example, let's say that you have a webpage at `https://foo.com` containing some front-end JavaScript code. If this JavaScript tries to make an HTTP request to `https://bar.com/data.json` (a different origin), then the request will be sent and processed by the `bar.com` server, but the user's web browser will *block the response* so that the JavaScript code from `https://foo.com` cannot see it.

Generally speaking, the same-origin policy is an extremely useful security safeguard. But while it's good in the general case, in certain circumstances you might want to relax it.

For example, if you have an API at `api.example.com` and a trusted JavaScript front-end application running on `www.example.com`, then you'll probably want to allow cross-origin requests from the trusted `www.example.com` domain to your API.

Or perhaps you have a completely open public API, and you want to allow cross-origin requests *from anywhere* so it's easy for other developers to integrate with their own websites.

Fortunately, most modern web browsers allow you to allow or disallow specific cross-origin requests to your API by setting `Access-Control` headers on your API responses. We'll explain exactly how to do that and how these headers work over the next few chapters of this book.

Demonstrating the Same-Origin Policy

To demonstrate how the same-origin policy works and how to relax it for requests to our API, we need to simulate *a request to our API from a different origin*.

As we're already using Go in this book, let's quickly make a second, very simple, Go application to make this cross-origin request. Essentially, we want this second application to serve a webpage containing some JavaScript, which in turn makes a request to our `GET /v1/healthcheck` endpoint.

If you're following along, create a new `cmd/examples/cors/simple/main.go` file to hold the code for this second application:

```
$ mkdir -p cmd/examples/cors/simple
$ touch cmd/examples/cors/simple/main.go
```

And add the following content:

File: cmd/examples/cors/simple/main.go

```
package main

import (
    "flag"
    "log"
    "net/http"
)

// Define a string constant containing the HTML for the webpage. This consists of a <h1>
// header tag, and some JavaScript which fetches the JSON from our GET /v1/healthcheck
// endpoint and writes it to inside the <div id="output"></div> element.
const html = `
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
</head>
<body>
    <h1>Simple CORS</h1>
    <div id="output"></div>
    <script>
        document.addEventListener('DOMContentLoaded', function() {
            fetch("http://localhost:4000/v1/healthcheck").then(
                function (response) {
                    response.text().then(function (text) {
                        document.getElementById("output").innerHTML = text;
                    });
                },
                function(err) {
                    document.getElementById("output").innerHTML = err;
                }
            );
        });
    </script>
</body>
</html>`

func main() {
    // Make the server address configurable at runtime via a command-line flag.
    addr := flag.String("addr", ":9000", "Server address")
    flag.Parse()

    log.Printf("starting server on %s", *addr)

    // Start a HTTP server listening on the given address, which responds to all
    // requests with the webpage HTML above.
    err := http.ListenAndServe(*addr, http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        w.Write([]byte(html))
    }))
    log.Fatal(err)
}
```

The Go code here should be nice and familiar to you already, but let's take a closer look at the JavaScript code in the `<script>` tag and explain what it does.

```

<script>
  document.addEventListener('DOMContentLoaded', function() {
    fetch("http://localhost:4000/v1/healthcheck").then(
      function (response) {
        response.text().then(function (text) {
          document.getElementById("output").innerHTML = text;
        });
      },
      function(err) {
        document.getElementById("output").innerHTML = err;
      }
    );
  });
</script>

```

In this code:

- We use the `fetch()` function to make a request to our API healthcheck endpoint. By default this sends a `GET` request, but it's also possible to configure this to use different HTTP methods and add custom headers. We'll explain how to do that later.
- The `fetch()` method works *asynchronously* and returns a `promise`. We use the `then()` method on the promise to set up two callback functions: the first callback is executed if `fetch()` is successful, and the second is executed if there is a failure.
- In our 'successful' callback we read the response body with `response.text()`, and use `document.getElementById("output").innerHTML` to replace the contents of the `<div id="output"></div>` element with this response body.
- In the 'failure' callback we replace the contents of the `<div id="output"></div>` element with the error message.
- This logic is all wrapped up in `document.addEventListener('DOMContentLoaded', function(){...})`, which essentially means that `fetch()` won't be called until the user's web browser has completely loaded the HTML document.

Note: I've explained the above because it's interesting, but this book isn't about JavaScript and you don't really need to worry about the details here. All you need to know is that *the JavaScript makes a HTTP request to our API's healthcheck endpoint, and then dumps the response inside the `<div id="output"></div>` element.*

Demonstration

OK, let's try this out. Go ahead and start up the new application:

```
$ go run ./cmd/examples/cors/simple
2021/04/17 17:23:14 starting server on :9000
```

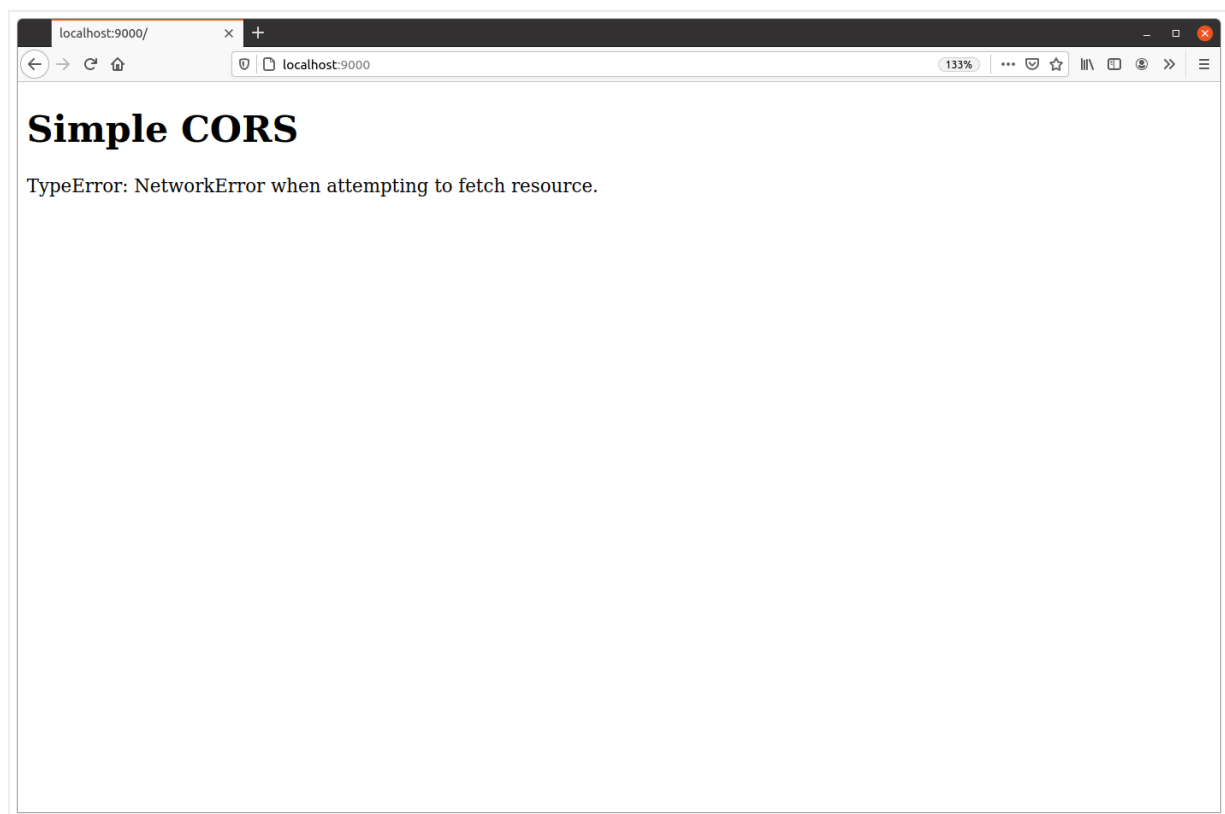
And then open a second terminal window and start our regular API application at the same time:

```
$ go run ./cmd/api
{"level":"INFO","time":"2021-04-17T15:23:34Z","message":"database connection pool established"}
{"level":"INFO","time":"2021-04-17T15:23:34Z","message":"starting server","properties":{"addr":":4000","env":"development"}}
```

At this point, you should now have the API running on the origin <http://localhost:4000> and the webpage with the JavaScript running on the origin <http://localhost:9000>. Because the ports are different, these are two different origins.

So, when you visit <http://localhost:9000> in your web browser, the `fetch()` action to <http://localhost:4000/v1/healthcheck> should be forbidden by the same-origin policy. Specifically, our API should receive and process the request, but your web browser should block the response from being read by the JavaScript code.

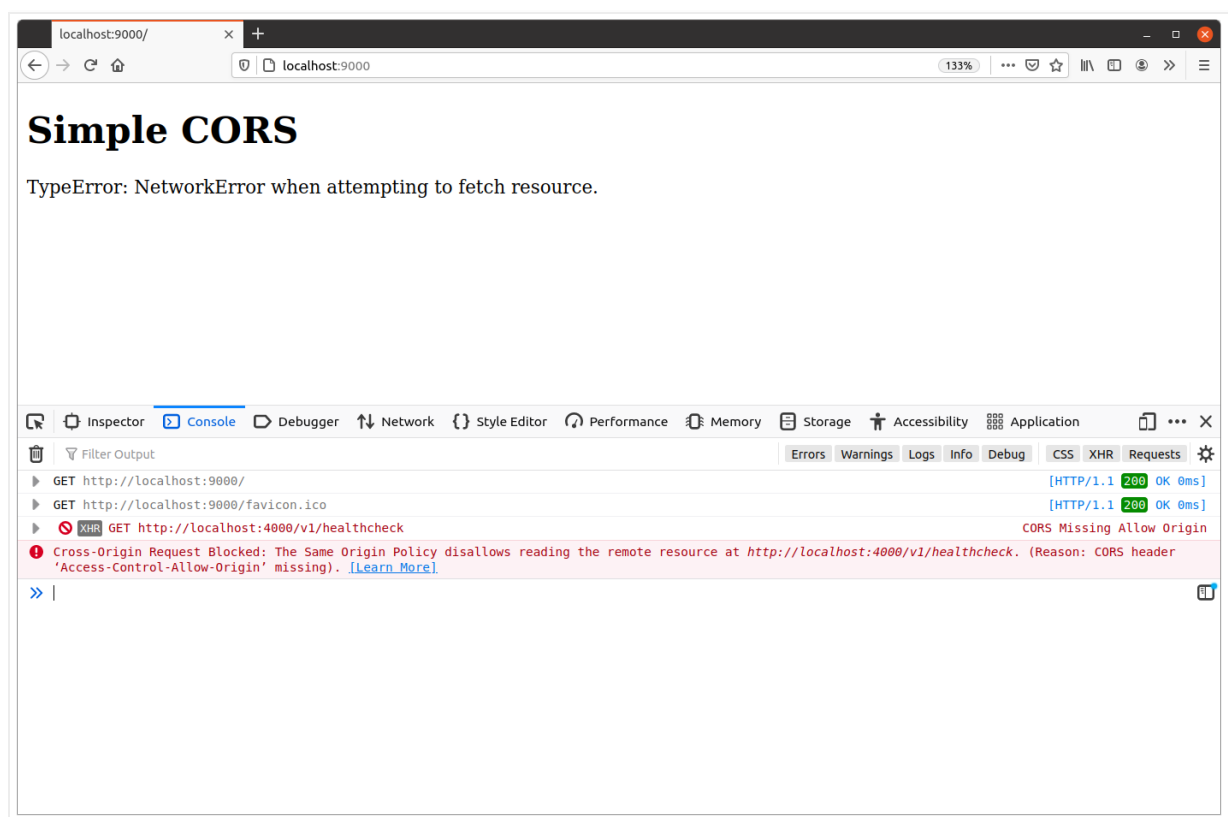
Let's take a look. If you open your web browser and visit <http://localhost:9000>, you should see the *Simple CORS* header followed by an error message similar to this:



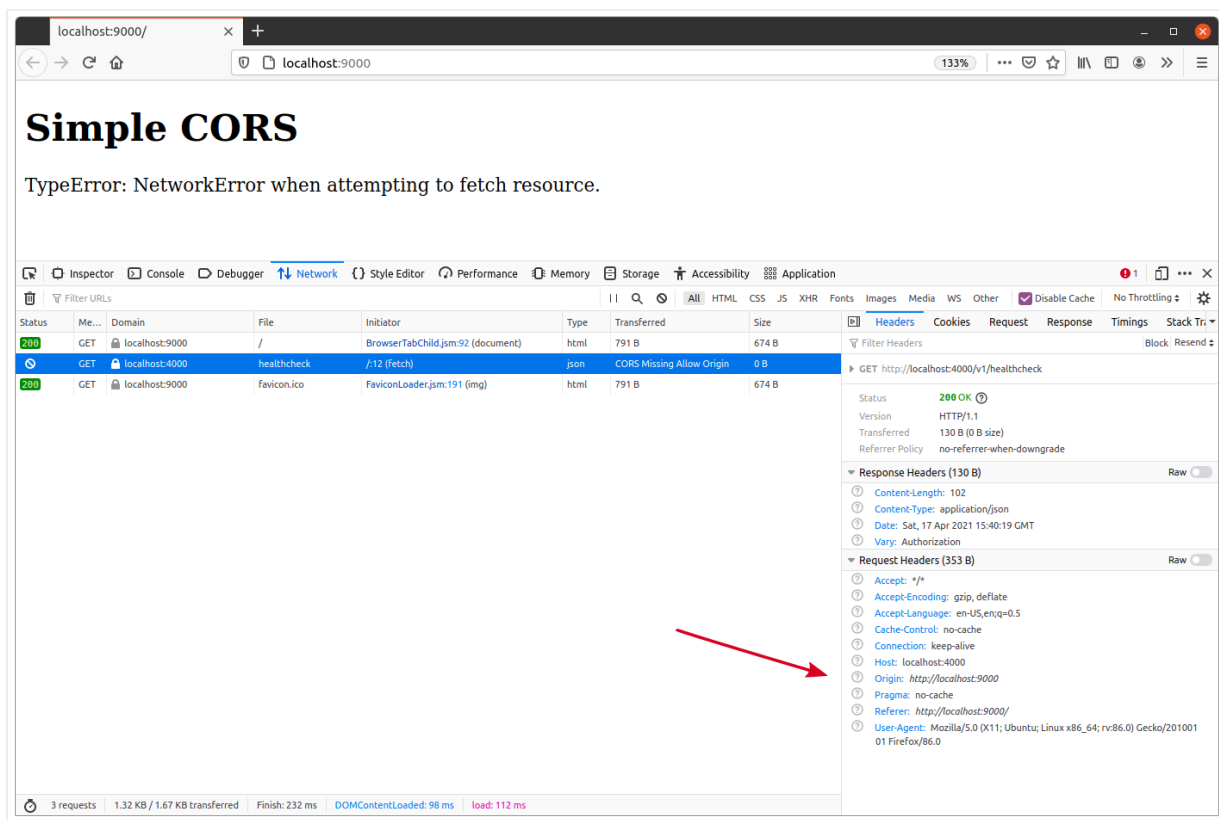
Note: The error message you see here is defined by your web browser, so yours might be slightly different if you're not using Firefox.

It's also helpful here to open your browser's developer tools, refresh the page, and look at your console log. You should see a message stating that the response from our `GET /v1/healthcheck` endpoint was blocked from being read due to the same-origin policy, similar to this:

Cross-Origin Request Blocked: The Same Origin Policy disallows reading the remote resource at `http://localhost:4000/v1/healthcheck`.



You may also want to open the network activity tab in your developer tools and examine the HTTP headers associated with the blocked request.



There are a couple of important things to point out here.

The first thing is that the headers demonstrate that the request was sent to our API, which processed the request and returned a successful **200 OK** response to the web browser containing all our standard response headers. To re-iterate: *the request itself was not prevented by the same-origin policy — it's just that the browser won't let JavaScript see the response.*

The second thing is that the web browser automatically set an **Origin** header on the request to show where the request originates from (highlighted by the red arrow above). You'll see that the header looks like this:

```
Origin: http://localhost:9000
```

We'll use this header in the next chapter to help us selectively relax the same-origin policy, depending on whether we trust the origin that a request is coming from.

Finally, it's important to emphasize that the same-origin policy is *a web browser thing only*.

Outside of a web browser, anyone can make a request to our API from anywhere, using **curl**, **wget** or any other means and read the response. That's completely unaffected and unchanged by the same-origin policy.

Simple CORS Requests

Let's now make some changes to our API which *relax the same-origin policy*, so that JavaScript can read the responses from our API endpoints.

To start with, the simplest way to achieve this is by setting the following header on all our API responses:

```
Access-Control-Allow-Origin: *
```

The **Access-Control-Allow-Origin** response header is used to indicate to a browser that it's OK to share a response with a different origin. In this case, the header value is the wildcard ***** character, which means that it's OK to share the response with *any other origin*.

Let's go ahead and create a small **enableCORS()** middleware function in our API application which sets this header:

```
File: cmd/api/middleware.go
```

```
package main

...

func (app *application) enableCORS(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        w.Header().Set("Access-Control-Allow-Origin", "*")

        next.ServeHTTP(w, r)
    })
}
```

And then update your **cmd/api/routes.go** file so that this middleware is used on all the application routes. Like so:

File: cmd/api/routes.go

```
package main

...

func (app *application) routes() http.Handler {
    router := httprouter.New()

    router.NotFound = http.HandlerFunc(app.notFoundResponse)
    router.MethodNotAllowed = http.HandlerFunc(app.methodNotAllowedResponse)

    router.HandleFunc(http.MethodGet, "/v1/healthcheck", app.healthcheckHandler)

    router.HandleFunc(http.MethodGet, "/v1/movies", app.requirePermission("movies:read", app.listMoviesHandler))
    router.HandleFunc(http.MethodPost, "/v1/movies", app.requirePermission("movies:write", app.createMovieHandler))
    router.HandleFunc(http.MethodGet, "/v1/movies/:id", app.requirePermission("movies:read", app.showMovieHandler))
    router.HandleFunc(http.MethodPatch, "/v1/movies/:id", app.requirePermission("movies:write", app.updateMovieHandler))
    router.HandleFunc(http.MethodDelete, "/v1/movies/:id", app.requirePermission("movies:write", app.deleteMovieHandler))

    router.HandleFunc(http.MethodPost, "/v1/users", app.registerUserHandler)
    router.HandleFunc(http.MethodPut, "/v1/users/activated", app.activateUserHandler)

    router.HandleFunc(http.MethodPost, "/v1/tokens/authentication", app.createAuthenticationTokenHandler)

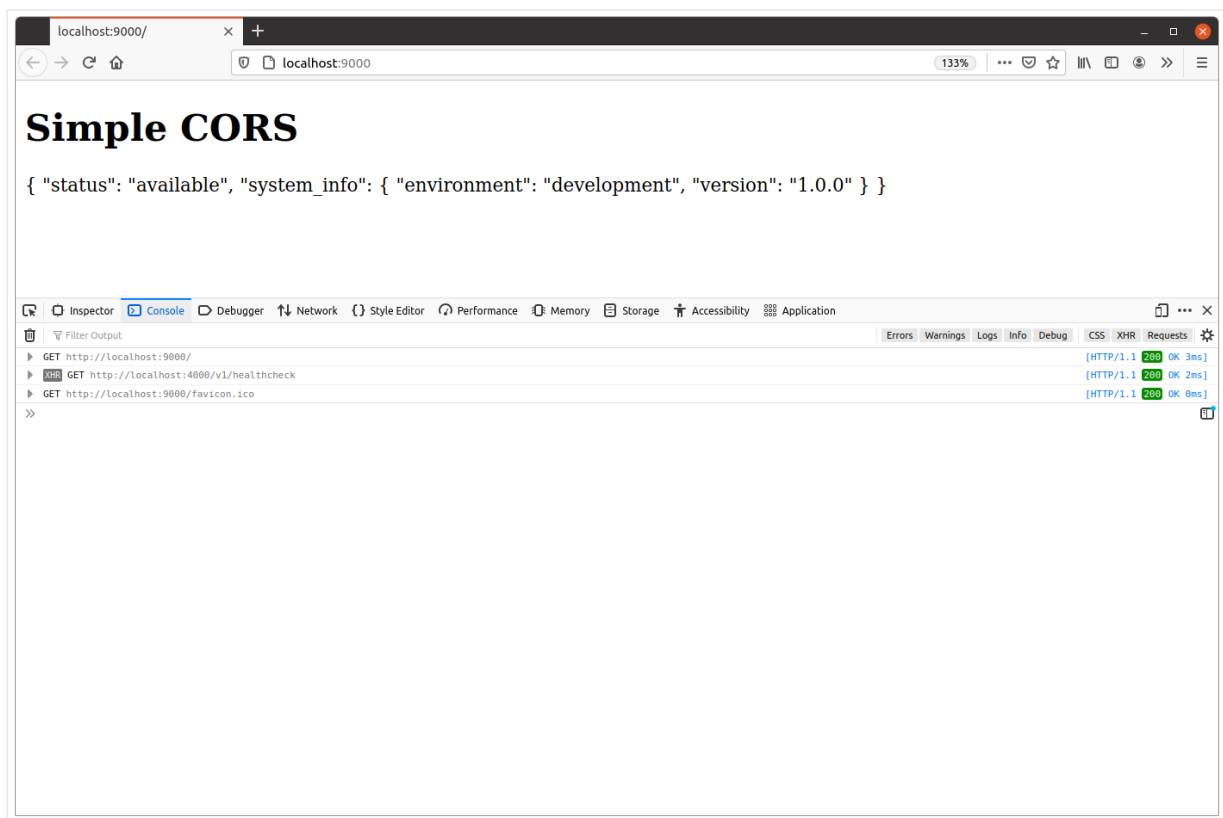
    // Add the enableCORS() middleware.
    return app.recoverPanic(app.enableCORS(app.rateLimit(app.authenticate(router))))
}
```

It's important to point out here that the `enableCORS()` middleware is deliberately positioned early in the middleware chain.

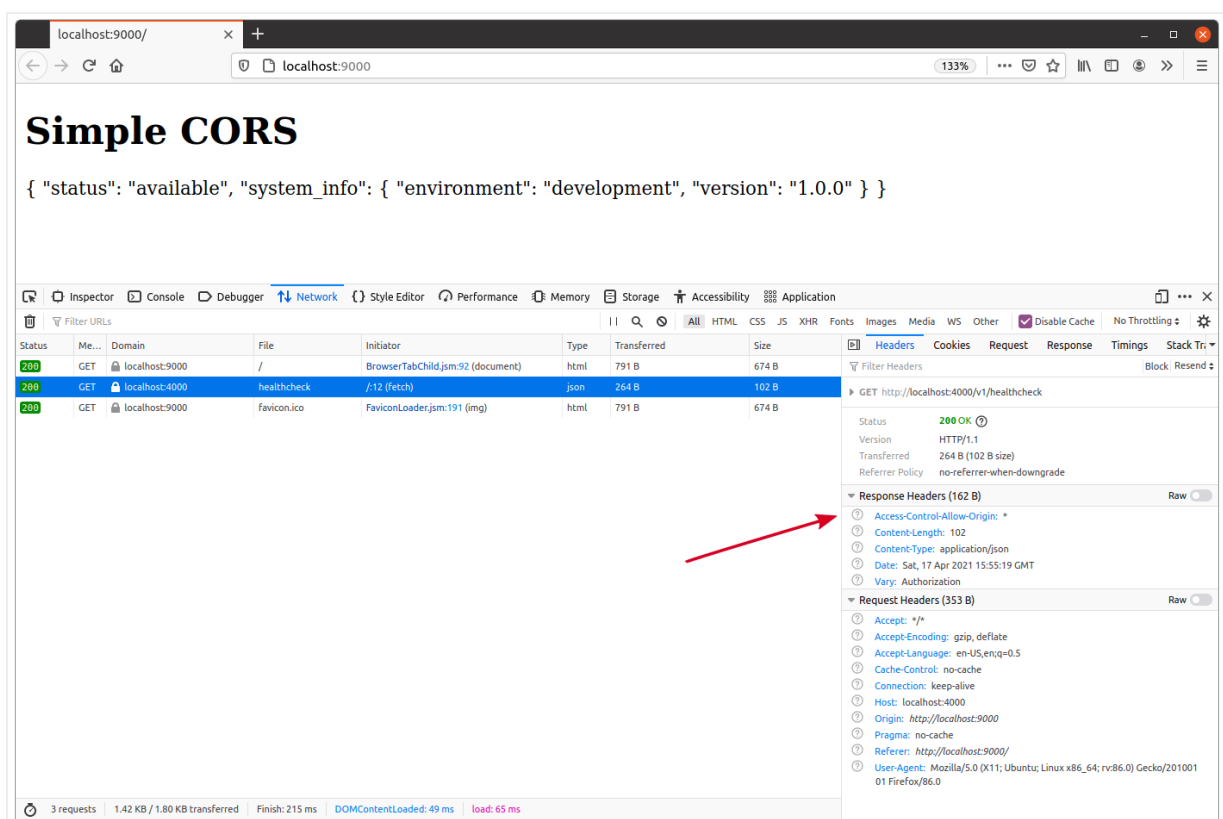
If we positioned it after our rate limiter, for example, any cross-origin requests that exceed the rate limit would *not have the Access-Control-Allow-Origin header set*. This means that they would be blocked by the client's web browser due to the same-origin policy, rather than the client receiving a **429 Too Many Requests** response like they should.

OK, let's try this out.

Restart the API application and then try visiting `http://localhost:9000` in your browser again. This time the cross-origin request should complete successfully, and you should see the JSON from our healthcheck handler presented on the webpage. Like so:



I also recommend taking a quick look at the request and response headers relating to the JavaScript `fetch()` request again. You should see that the `Access-Control-Allow-Origin: *` header has been set on the response, similar to this:



Restricting Origins

Using a wildcard to allow cross-origin requests, like we are in the code above, can be useful in certain circumstances (like when you have a completely public API with no access control checks). But more often you'll probably want to restrict CORS to a much smaller set of *trusted origins*.

To do this, you need to explicitly include the trusted origins in the `Access-Control-Allow-Origin` header instead of using a wildcard. For example, if you want to only allow CORS from the origin `https://www.example.com` you could send the following header in your responses:

```
Access-Control-Allow-Origin: https://www.example.com
```

If you only have one, fixed, origin that you want to allow requests from, then doing this is quite simple — you can just update your `enableCORS()` middleware to hard-code in the necessary origin value.

But if you need to support multiple trusted origins, or you want the value to be configurable at runtime, then things get a bit more complex.

One of the problems is that — in practice — you can only specify *exactly one origin* in the `Access-Control-Allow-Origin` header. You can't include a list of multiple origin values, separated by spaces or commas like you might expect.

To work around this limitation, you'll need to update your `enableCORS()` middleware to check if the value of the `Origin` header matches one of your trusted origins. If it does, then you can reflect (or *echo*) that value back in the `Access-Control-Allow-Origin` response header.

Note: The [web origin specification](#) does permit multiple *space-separated* values in the `Access-Control-Allow-Origin` header but, unfortunately, no web browsers actually support this.

Supporting multiple dynamic origins

Let's update our API so that cross-origin requests are restricted to a list of trusted origins, configurable at runtime.

The first thing we'll do is add a new `-cors-trusted-origins` command-line flag to our API application, which we can use to specify the list of trusted origins at runtime. We'll set this up so that the origins must be separated by a space character — like so:

```
$ go run ./cmd/api -cors-trusted-origins="https://www.example.com https://staging.example.com"
```

In order to process this command-line flag, we can combine the `flags.Func()` and `strings.Fields()` functions to split the origin values into a `[]string` slice ready for use.

If you're following along, open your `cmd/api/main.go` file and add the following code:

```
File: cmd/api/main.go

package main

import (
    "context"
    "database/sql"
    "flag"
    "os"
    "strings" // New import
    "sync"
    "time"

    "greenlight.alexedwards.net/internal/data"
    "greenlight.alexedwards.net/internal/jsonlog"
    "greenlight.alexedwards.net/internal/mailer"

    _ "github.com/lib/pq"
)

const version = "1.0.0"

type config struct {
    port int
    env string
    db struct {
        dsn string
        maxOpenConns int
        maxIdleConns int
        maxIdleTime string
    }
    limiter struct {
        enabled bool
        rps float64
        burst int
    }
    smtp struct {
        host string
        port int
        username string
        password string
        sender string
    }
}

// Add a cors struct and trustedOrigins field with the type []string.
cors struct {
    trustedOrigins []string
}
```

```

}

...

func main() {
    var cfg config

    flag.IntVar(&cfg.port, "port", 4000, "API server port")
    flag.StringVar(&cfg.env, "env", "development", "Environment (development|staging|production)")

    flag.StringVar(&cfg.db.dsn, "db-dsn", os.Getenv("GREENLIGHT_DB_DSN"), "PostgreSQL DSN")

    flag.IntVar(&cfg.db.maxOpenConns, "db-max-open-conns", 25, "PostgreSQL max open connections")
    flag.IntVar(&cfg.db.maxIdleConns, "db-max-idle-conns", 25, "PostgreSQL max idle connections")
    flag.StringVar(&cfg.db.maxIdleTime, "db-max-idle-time", "15m", "PostgreSQL max connection idle time")

    flag.BoolVar(&cfg.limiter.enabled, "limiter-enabled", true, "Enable rate limiter")
    flag.Float64Var(&cfg.limiter.rps, "limiter-rps", 2, "Rate limiter maximum requests per second")
    flag.IntVar(&cfg.limiter.burst, "limiter-burst", 4, "Rate limiter maximum burst")

    flag.StringVar(&cfg.smtp.host, "smtp-host", "smtp.mailtrap.io", "SMTP host")
    flag.IntVar(&cfg.smtp.port, "smtp-port", 25, "SMTP port")
    flag.StringVar(&cfg.smtp.username, "smtp-username", "0abf276416b183", "SMTP username")
    flag.StringVar(&cfg.smtp.password, "smtp-password", "d8672aa2264bb5", "SMTP password")
    flag.StringVar(&cfg.smtp.sender, "smtp-sender", "Greenlight <no-reply@greenlight.alexedwards.net>", "SMTP sender")

    // Use the flag.Func() function to process the -cors-trusted-origins command line
    // flag. In this we use the strings.Fields() function to split the flag value into a
    // slice based on whitespace characters and assign it to our config struct.
    // Importantly, if the -cors-trusted-origins flag is not present, contains the empty
    // string, or contains only whitespace, then strings.Fields() will return an empty
    // []string slice.
    flag.Func("cors-trusted-origins", "Trusted CORS origins (space separated)", func(val string) error {
        cfg.cors.trustedOrigins = strings.Fields(val)
        return nil
    })

    flag.Parse()

    ...
}

...

```

Note: If you'd like to find out more about the new `flag.Func()` functionality and some of the ways in which you can use it, I've written a much more detailed blog post about it [here](#).

Once that's done, the next step is to update our `enableCORS()` middleware. Specifically, we want the middleware to check if the value of the request `Origin` header is an exact, case-sensitive, match for one of our trusted origins. If there is a match, then we should set an `Access-Control-Allow-Origin` response header which reflects (or echoes) back the value of the request's `Origin` header.

Otherwise, we should allow the request to proceed as normal *without* setting an

Access-Control-Allow-Origin response header. In turn, that means that any cross-origin responses will be blocked by a web browser, just like they were originally.

A side effect of this is that the *response will be different depending on the origin that the request is coming from*. Specifically, the value of the **Access-Control-Allow-Origin** header may be different in the response, or it may not even be included at all.

So because of this we should make sure to always set a **Vary: Origin** response header to warn any caches that the response may be different. This is actually really important, and it can be the cause of subtle bugs [like this one](#) if you forget to do it. As a rule of thumb:

*If your code makes a decision about what to return based on the content of a request header, you should include that header name in your **Vary** response header — even if the request didn't include that header.*

OK, let's update our **enableCORS()** middleware in line with the logic above, like so:

File: cmd/api/middleware.go

```
package main

...

func (app *application) enableCORS(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        // Add the "Vary: Origin" header.
        w.Header().Add("Vary", "Origin")

        // Get the value of the request's Origin header.
        origin := r.Header.Get("Origin")

        // Only run this if there's an Origin request header present.
        if origin != "" {
            // Loop through the list of trusted origins, checking to see if the request
            // origin exactly matches one of them. If there are no trusted origins, then
            // the loop won't be iterated.
            for i := range app.config.cors.trustedOrigins {
                if origin == app.config.cors.trustedOrigins[i] {
                    // If there is a match, then set a "Access-Control-Allow-Origin"
                    // response header with the request origin as the value and break
                    // out of the loop.
                    w.Header().Set("Access-Control-Allow-Origin", origin)
                    break
                }
            }
        }

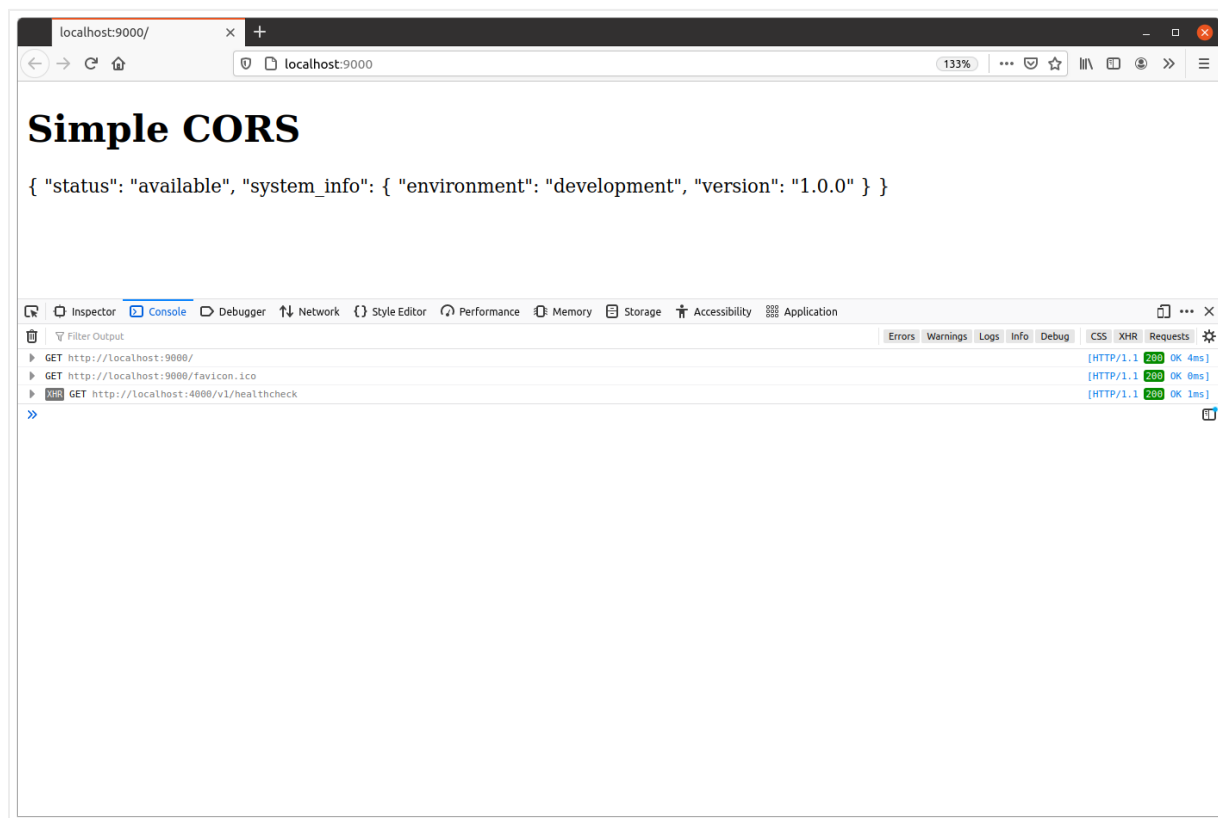
        // Call the next handler in the chain.
        next.ServeHTTP(w, r)
    })
}
```

And with those changes complete, we're now ready to try this out again.

Restart your API, passing in `http://localhost:9000` and `http://localhost:9001` as trusted origins like so:

```
$ go run ./cmd/api -cors-trusted-origins="http://localhost:9000 http://localhost:9001"
{"level":"INFO","time":"2021-04-17T16:20:49Z","message":"database connection pool established"}
{"level":"INFO","time":"2021-04-17T16:20:49Z","message":"starting server","properties":{"addr":":4000","env":"development"}}
```

And when you refresh `http://localhost:9000` in your browser, you should find that the cross-origin request still works successfully.

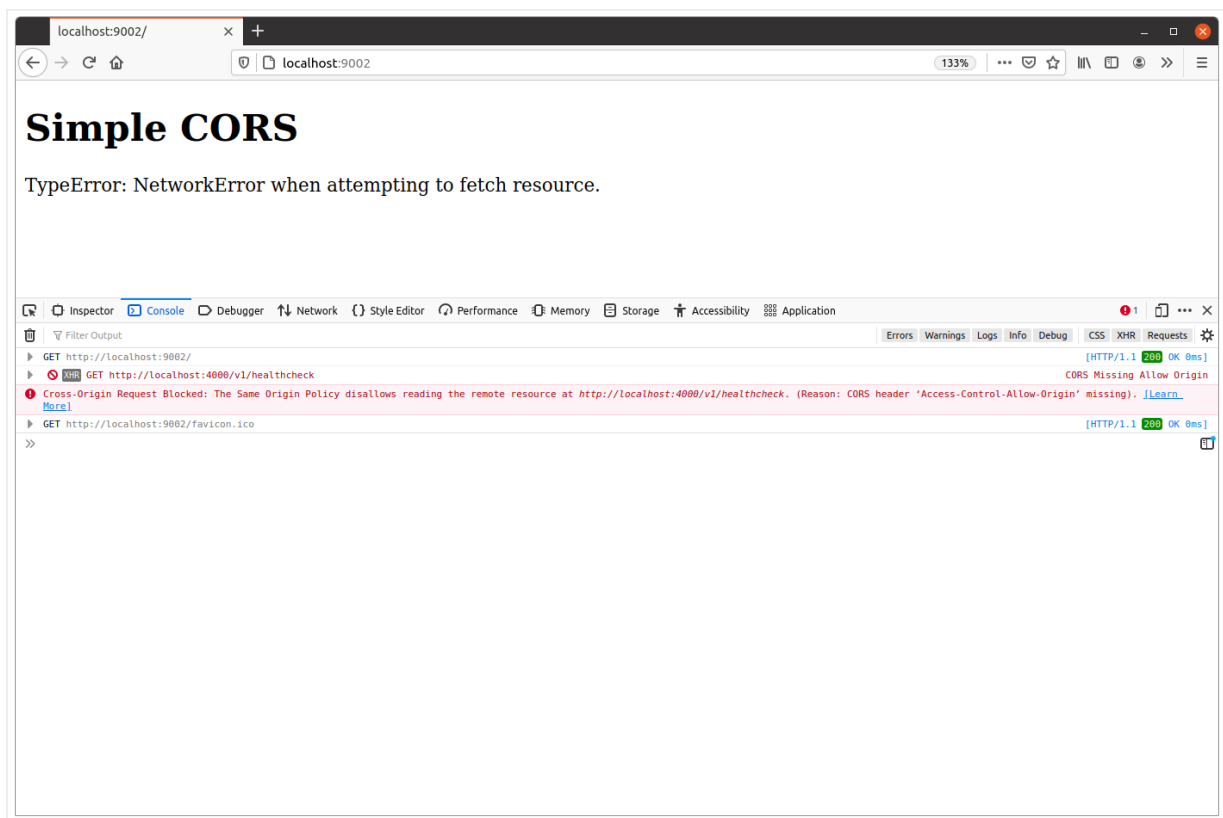


If you want, you can also try running the `cmd/examples/cors/simple` application with `:9001` as the server address, and you should find that the cross-origin request works from that too.

In contrast, try running the `cmd/examples/cors/simple` application with the address `:9002`.

```
$ go run ./cmd/examples/cors/simple --addr=":9002"
2021/04/17 18:24:22 starting server on :9002
```

This will give the webpage an origin of `http://localhost:9002` — which isn't one of our trusted origins — so when you visit `http://localhost:9002` in your browser you should find that the cross-origin request is blocked. Like so:



Additional Information

Partial origin matches

If you have a lot of trusted origins that you want to support, then you might be tempted to check for a partial match on the origin to see if it ‘starts with’ or ‘ends with’ a specific value, or matches a regular expression. If you do this, you must take a lot of care to avoid any unintentional matches.

As a simple example, if `http://example.com` and `http://www.example.com` are your trusted origins, your first thought might check that the request `Origin` header ends with `example.com`. This would be a bad idea, as an attacker could register the domain name `attackerexample.com` and any requests from that origin would pass your check.

This is just one simple example — and the following blog posts discuss some of the other vulnerabilities that can arise when using partial match or regular expression checks:

- [Security Risks of CORS](#)
- [Exploiting CORS misconfigurations for Bitcoins and bounties](#)

Generally, it's best to check the **Origin** request header against an explicit safelist of full-length trusted origins, like we have done in this chapter.

The null origin

It's important to never include the value **"null"** as a trusted origin in your safelist. This is because the request header **Origin: null** can be forged by an attacker by sending a request from a [sandboxed iframe](#).

Authentication and CORS

If your API endpoint requires *credentials* (cookies or HTTP basic authentication) you should also set an **Access-Control-Allow-Credentials: true** header in your responses. If you don't set this header, then the web browser will prevent any cross-origin responses with credentials from being read by JavaScript.

Importantly, you must never use the wildcard **Access-Control-Allow-Origin: *** header in conjunction with **Access-Control-Allow-Credentials: true**, as this would allow any website to make a credentialed cross-origin request to your API.

Also, importantly, if you want credentials to be sent with a cross-origin request then you'll need to explicitly specify this in your JavaScript. For example, with **fetch()** you should set the **credentials** value of the request to **'include'**. Like so:

```
fetch("https://api.example.com", {credentials: 'include'}).then( ... );
```

Or if using **XMLHttpRequest** you should set the **withCredentials** property to **true**. For example:

```
var xhr = new XMLHttpRequest();
xhr.open('GET', 'https://api.example.com');
xhr.withCredentials = true;
xhr.send(null);
```

Preflight CORS Requests

The cross-origin request that we made from JavaScript in the previous chapter is known as a *simple* cross-origin request. Broadly speaking, cross-origin requests are classified as ‘simple’ when *all* the following conditions are met:

- The request HTTP method is one of the three CORS-safe methods: `HEAD`, `GET` or `POST`.
- The request headers are all either `forbidden headers` or one of the four CORS-safe headers:
 - `Accept`
 - `Accept-Language`
 - `Content-Language`
 - `Content-Type`
- The value for the `Content-Type` header (if set) is one of:
 - `application/x-www-form-urlencoded`
 - `multipart/form-data`
 - `text/plain`

When a cross-origin request doesn’t meet these conditions, then the web browser will trigger an initial ‘preflight’ request *before the real request*. The purpose of this preflight request is to determine whether the *real* cross-origin request will be permitted or not.

Demonstrating a preflight request

To help demonstrate how preflight requests work and what we need to do to deal with them, let’s create another example webpage under the `cmd/examples/cors/` directory.

We’ll set up this webpage so it makes a request to our `POST /v1/tokens/authentication` endpoint. When calling this endpoint we’ll include an email address and password in a JSON request body, along with a `Content-Type: application/json` header. And because the header `Content-Type: application/json` isn’t allowed in a ‘simple’ cross-origin request, this should trigger a preflight request to our API.

Go ahead and create a new file at `cmd/examples/cors/preflight/main.go`:

```
$ mkdir -p cmd/examples/cors/preflight  
$ touch cmd/examples/cors/preflight/main.go
```

And add the code below, which follows a very similar pattern to the one we used a couple of chapters ago.

File: cmd/examples/cors/preflight/main.go

```
package main

import (
    "flag"
    "log"
    "net/http"
)

// Define a string constant containing the HTML for the webpage. This consists of a <h1>
// header tag, and some JavaScript which calls our POST /v1/tokens/authentication
// endpoint and writes the response body to inside the <div id="output"></div> tag.
const html = `
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
</head>
<body>
    <h1>Preflight CORS</h1>
    <div id="output"></div>
    <script>
        document.addEventListener('DOMContentLoaded', function() {
            fetch("http://localhost:4000/v1/tokens/authentication", {
                method: "POST",
                headers: {
                    'Content-Type': 'application/json'
                },
                body: JSON.stringify({
                    email: 'alice@example.com',
                    password: 'pa55word'
                })
            }).then(
                function (response) {
                    response.text().then(function (text) {
                        document.getElementById("output").innerHTML = text;
                    });
                },
                function(err) {
                    document.getElementById("output").innerHTML = err;
                }
            );
        });
    </script>
</body>
</html>`

func main() {
    addr := flag.String("addr", ":9000", "Server address")
    flag.Parse()

    log.Printf("starting server on %s", *addr)

    err := http.ListenAndServe(*addr, http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        w.Write([]byte(html))
    })))
    log.Fatal(err)
}
```

If you're following along, go ahead and run this application:

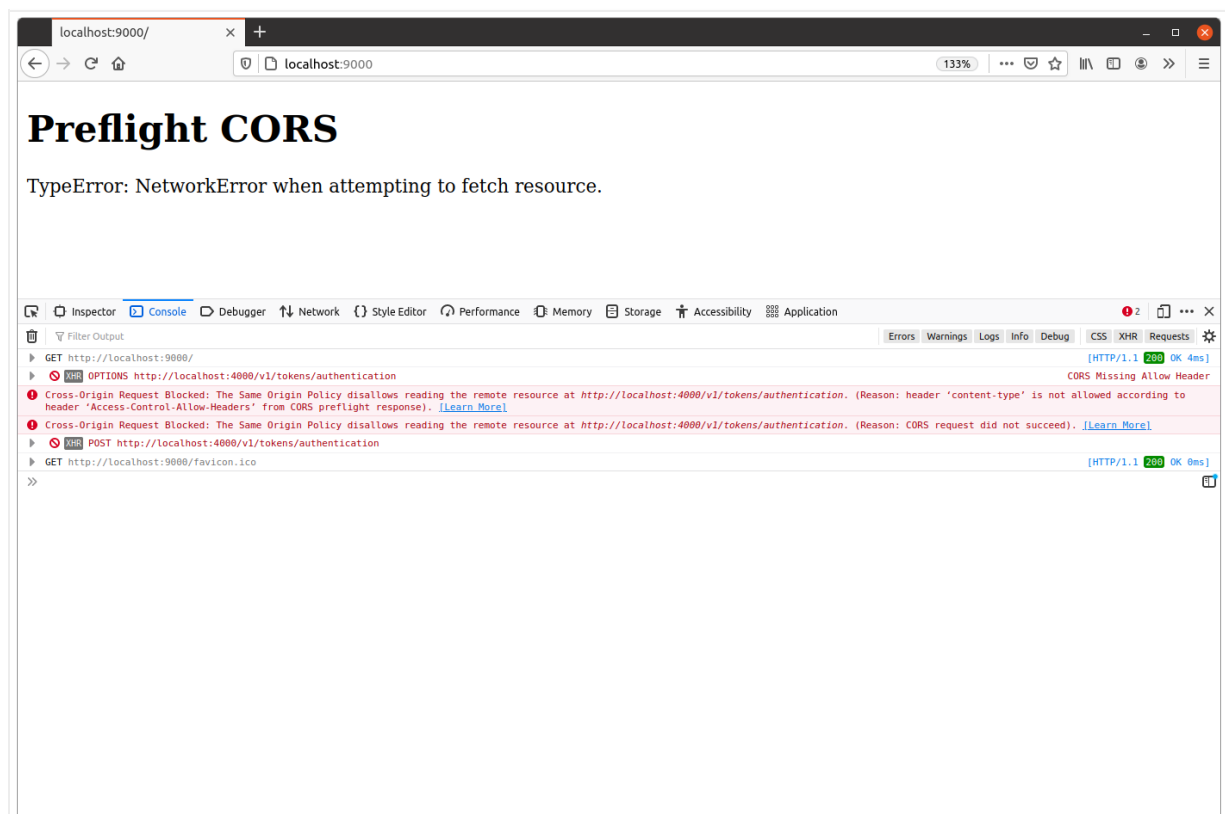
```
$ go run ./cmd/examples/cors/preflight
2021/04/17 18:47:55 starting server on :9000
```

Then open a second terminal window and start our regular API application at the same time with `http://localhost:9000` as a trusted origin:

```
$ go run ./cmd/api -cors-trusted-origins="http://localhost:9000"
{"level":"INFO","time":"2021-04-17T16:48:55Z","message":"database connection pool established"}
{"level":"INFO","time":"2021-04-17T16:48:55Z","message":"starting server","properties":{"addr":":4000","env":"development"}}
```

Once both are running, open your web browser and navigate to `http://localhost:9000`. If you look at the console log in your developer tools, you should see a message similar to this:

Cross-Origin Request Blocked: The Same Origin Policy disallows reading the remote resource at http://localhost:4000/v1/tokens/authentication. (Reason: header 'content-type' is not allowed according to header 'Access-Control-Allow-Headers' from CORS preflight response).

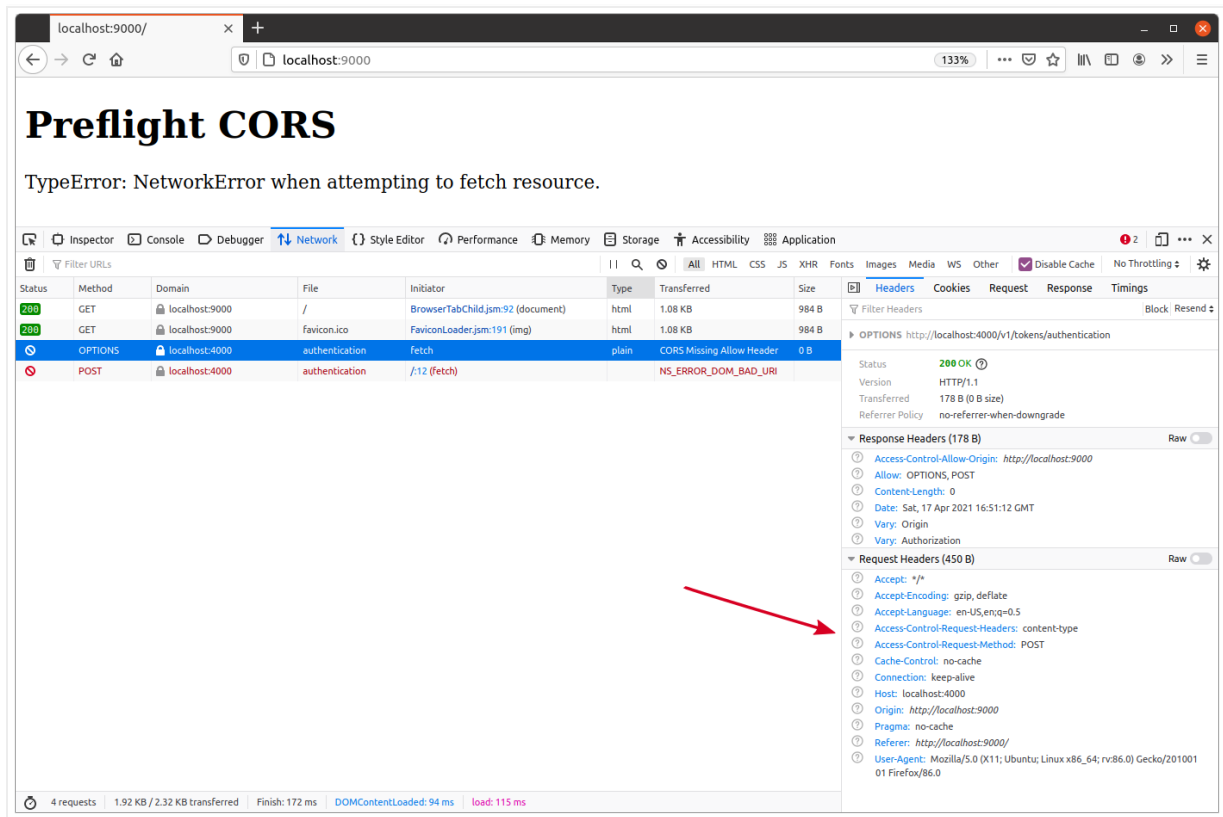


We can see that there are two requests here marked as ‘blocked’ by the browser:

- An `OPTIONS /v1/tokens/authentication` request (this is the preflight request).

- A **POST** `/v1/tokens/authentication` request (this is the ‘real’ request).

Let’s take a closer look at the preflight request in the network tab of the developer tools:



The interesting thing here is the preflight request headers. They might look slightly different for you depending on the browser you’re using, but broadly they should look something like this:

```
Accept: */*
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.5
Access-Control-Request-Headers: content-type
Access-Control-Request-Method: POST
Cache-Control: no-cache
Connection: keep-alive
Host: localhost:4000
Origin: http://localhost:9000
Pragma: no-cache
Referer: http://localhost:9000/
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:86.0) Gecko/20100101 Firefox/86.0
```

There are three headers here which are relevant to CORS:

- **Origin** — As we saw previously, this lets our API know what origin the preflight request is coming from.
- **Access-Control-Request-Method** — This lets our API know what HTTP method will be

used for the real request (in this case, we can see that the real request will be a **POST**).

- **Access-Control-Request-Headers** — This lets our API know what HTTP headers will be sent with the real request (in this case we can see that the real request will include a **content-type** header).

It's important to note that **Access-Control-Request-Headers** won't list *all* the headers that the real request will use. Only headers that are *not* **CORS-safe** or **forbidden** will be listed. If there are no such headers, then **Access-Control-Request-Headers** may be omitted from the preflight request entirely.

Responding to preflight requests

In order to respond to a preflight request, the first thing we need to do is identify that it *is* a preflight request — rather than just a regular (possibly even cross-origin) **OPTIONS** request.

To do that, we can leverage the fact that preflight requests always have three components: the HTTP method **OPTIONS**, an **Origin** header, and an **Access-Control-Request-Method** header. If any one of these pieces is missing, we know that it is not a preflight request.

Once we identify that it is a preflight request, we need to send a **200 OK** response with some special headers to let the browser know whether or not it's OK for the real request to proceed. These are:

- An **Access-Control-Allow-Origin** response header, which reflects the value of the preflight request's **Origin** header (just like in the previous chapter).
- An **Access-Control-Allow-Methods** header listing the HTTP methods that can be used in real cross-origin requests to the URL.
- An **Access-Control-Allow-Headers** header listing the request headers that can be included in real cross-origin requests to the URL.

In our case, we could set the following response headers to allow cross-origin requests for *all our endpoints*:

```
Access-Control-Allow-Origin: <reflected trusted origin>
Access-Control-Allow-Methods: OPTIONS, PUT, PATCH, DELETE
Access-Control-Allow-Headers: Authorization, Content-Type
```

Important: When responding to a preflight request it's not necessary to include the CORS-safe methods `HEAD`, `GET` or `POST` in the `Access-Control-Allow-Methods` header. Likewise, it's not necessary to include forbidden or CORS-safe headers in `Access-Control-Allow-Headers`.

When the web browser receives these headers, it compares the values to the method and (case-insensitive) headers that it wants to use in the real request. If the method or any of the headers are not allowed, then the browser will block the real request.

Updating our middleware

Let's put this into action and update our `enableCORS()` middleware so it intercepts and responds to any preflight requests. Specifically, we want to:

1. Set a `Vary: Access-Control-Request-Method` header on all responses, as the response will be different depending on whether or not this header exists in the request.
2. Check whether the request is a preflight cross-origin request or not. If it's not, then we should allow the request to proceed as normal.
3. Otherwise, if it is a preflight cross-origin request, then we should add the `Access-Control-Allow-Method` and `Access-Control-Allow-Headers` headers as described above.

Go ahead and update the `cmd/api/middleware.go` file like so:

File: cmd/api/middleware.go

```
package main

...

func (app *application) enableCORS(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        w.Header().Add("Vary", "Origin")

        // Add the "Vary: Access-Control-Request-Method" header.
        w.Header().Add("Vary", "Access-Control-Request-Method")

        origin := r.Header.Get("Origin")

        if origin != "" {
            for i := range app.config.cors.trustedOrigins {
                if origin == app.config.cors.trustedOrigins[i] {
                    w.Header().Set("Access-Control-Allow-Origin", origin)

                    // Check if the request has the HTTP method OPTIONS and contains the
                    // "Access-Control-Request-Method" header. If it does, then we treat
                    // it as a preflight request.
                    if r.Method == http.MethodOptions && r.Header.Get("Access-Control-Request-Method") != "" {
                        // Set the necessary preflight response headers, as discussed
                        // previously.
                        w.Header().Set("Access-Control-Allow-Methods", "OPTIONS, PUT, PATCH, DELETE")
                        w.Header().Set("Access-Control-Allow-Headers", "Authorization, Content-Type")

                        // Write the headers along with a 200 OK status and return from
                        // the middleware with no further action.
                        w.WriteHeader(http.StatusOK)
                        return
                    }
                }
            }
            break
        }

        next.ServeHTTP(w, r)
    })
}
```

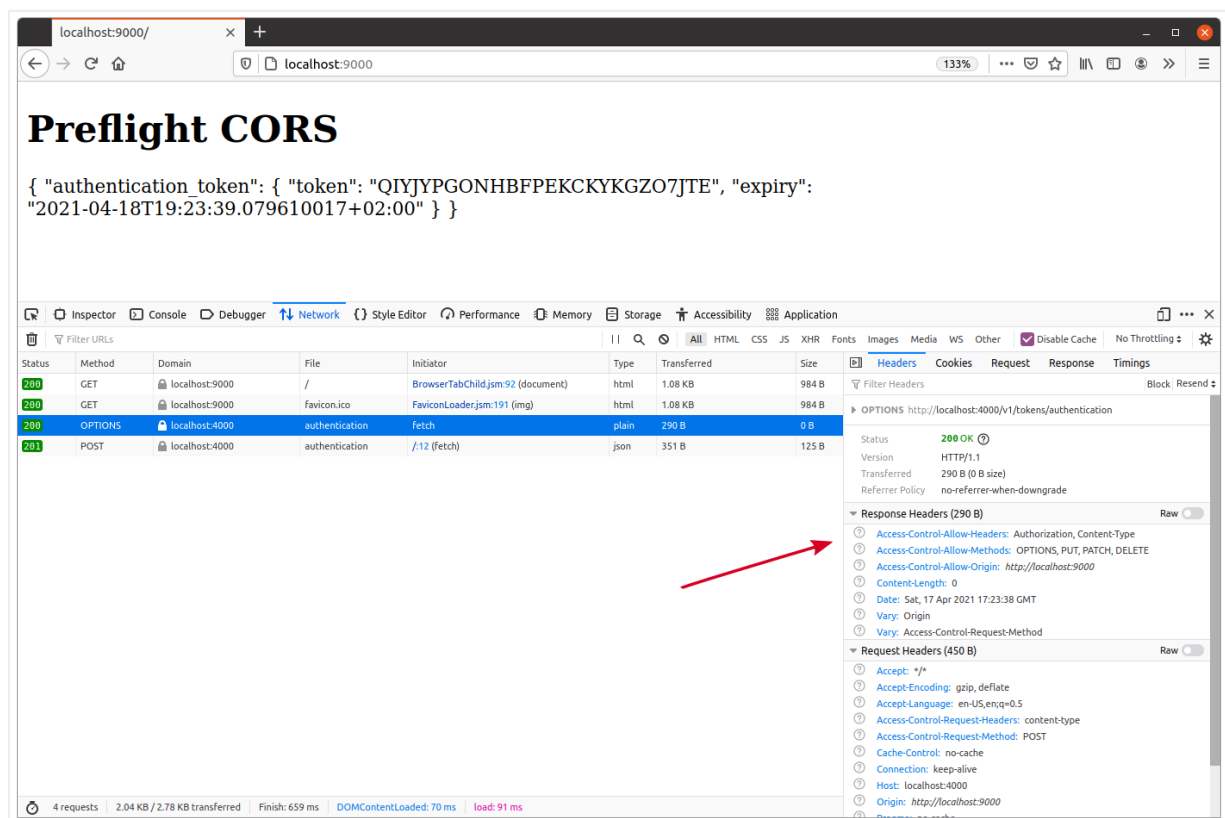
There are a couple of additional things to point out here:

- When we respond to a preflight request we deliberately send the HTTP status **200 OK** rather than **204 No Content** — even though there is no response body. This is because certain browser versions **may not support 204 No Content** responses and subsequently block the real request.
- If you allow the **Authorization** header in cross-origin requests, like we are in the code above, it's important to not set the wildcard **Access-Control-Allow-Origin: *** header or reflect the **Origin** header without checking against a list of trusted origins. Otherwise, this would leave your service vulnerable to a distributed brute-force attack against any authentication credentials that are passed in that header.

OK, let's try this out. Restart your API, again setting `http://localhost:9000` as a trusted origin like so:

```
$ go run ./cmd/api -cors-trusted-origins="http://localhost:9000"
{"level":"INFO","time":"2021-04-17T17:23:05Z","message":"database connection pool established"}
{"level":"INFO","time":"2021-04-17T17:23:05Z","message":"starting server","properties":{"addr":":4000","env":"development"}}
```

Then open `http://localhost:9000` in your browser again. This time you should see that the cross-origin `fetch()` to `POST /v1/tokens/authentication` succeeds, and you now get an authentication token in the response. Similar to this:



Note: If you look at the details for the preflight request, you should see that our new CORS headers have been set on the preflight response, shown by the red arrow in the screenshot above.

Additional Information

Caching preflight responses

If you want, you can also add an `Access-Control-Max-Age` header to your preflight responses. This indicates the number of seconds that the information provided by the `Access-Control-Allow-Methods` and `Access-Control-Allow-Headers` headers can be cached by the browser.

For example, to allow the values to be cached for 60 seconds you can set the following header on your preflight response:

```
Access-Control-Max-Age: 60
```

If you don't set an `Access-Control-Max-Age` header, current versions of Chrome/Chromium and Firefox will default to caching these preflight response values for 5 seconds. Older versions or other browsers may have different defaults, or not cache the values at all.

Setting a long `Access-Control-Max-Age` duration might seem like an appealing way to reduce requests to your API — and it is! But you also need to be careful. Not all browsers provide a way to clear the preflight cache, so if you send back the wrong headers the user will be stuck with them until the cache expires.

If you want to disable caching altogether, you can set the value to `-1`:

```
Access-Control-Max-Age: -1
```

It's also important to be aware that browsers may impose a hard maximum on how long the headers can be cached for. The [MDN documentation](#) says:

- *Firefox caps this at 24 hours (86400 seconds).*
- *Chromium (prior to v76) caps at 10 minutes (600 seconds).*
- *Chromium (starting in v76) caps at 2 hours (7200 seconds).*

Preflight wildcards

If you have a complex or rapidly changing API then it might be awkward to maintain a hard-coded safelist of methods and headers for the preflight response. You might think: *I just want to allow all HTTP methods and headers for cross-origin requests.*

In this case, both the `Access-Control-Allow-Methods` and `Access-Control-Allow-Headers` headers allow you to use a wildcard `*` character like so:

```
Access-Control-Allow-Methods: *  
Access-Control-Allow-Headers: *
```

But using these comes with some important caveats:

- Wildcards in these headers are currently only supported by [74% of browsers](#). Any browsers which don't support them will block the preflight request.
- The **Authorization** header cannot be wildcarded. Instead, you will need to include this explicitly in the header like **Access-Control-Allow-Headers: Authorization, ***.
- Wildcards are not supported for credentialed requests (those with cookies or HTTP basic authentication). For these, the character ***** will be treated as the literal string **"*"**, rather than as a wildcard.

Metrics

When your application is running in production with real-life traffic — or if you're carrying out targeted load testing — you might want some up-close insight into how it's performing and what resources it is using.

For example, you might want to answer questions like:

- How much memory is my application using? How is this changing over time?
- How many goroutines are currently in use? How is this changing over time?
- How many database connections are in use and how many are idle? Do I need to change the connection pool settings?
- What is the ratio of successful HTTP responses to both client and server errors? Are error rates elevated above normal?

Having insight into these things can help inform your hardware and configuration setting choices, and act as an early warning sign of potential problems (such as memory leaks).

To assist with this, Go's standard library includes the [expvar](#) package which makes it easy to collate and view different application metrics at runtime.

In this section you'll learn:

- How to use the [expvar](#) package to view application metrics in JSON format via a HTTP handler.
- What default application metrics are available, and how to create your own custom application metrics for monitoring the number of active goroutines and the database connection pool.
- How to use middleware to monitor request-level application metrics, including the counts of different HTTP status codes.

Exposing Metrics with Expvar

Viewing metrics for our API is made easy by the fact that the `expvar` package provides an `expvar.Handler()` function which returns a *HTTP handler exposing your application metrics*.

By default this handler displays information about memory usage, along with a reminder of what command-line flags you used when starting the application, all outputted in JSON format.

So the first thing that we're going to do is mount this handler at a new `GET /debug/vars` endpoint, like so:

Method	URL Pattern	Handler	Action
...	...	...	...
GET	/debug/vars	expvar.Handler()	Display application metrics

File: cmd/api/routes.go

```
package main

import (
    "expvar" // New import
    "net/http"

    "github.com/julienschmidt/httprouter"
)

func (app *application) routes() http.Handler {
    router := httprouter.New()

    router.NotFound = http.HandlerFunc(app.notFoundResponse)
    router.MethodNotAllowed = http.HandlerFunc(app.methodNotAllowedResponse)

    router.HandleFunc(http.MethodGet, "/v1/healthcheck", app.healthcheckHandler)

    router.HandleFunc(http.MethodGet, "/v1/movies", app.requirePermission("movies:read", app.listMoviesHandler))
    router.HandleFunc(http.MethodPost, "/v1/movies", app.requirePermission("movies:write", app.createMovieHandler))
    router.HandleFunc(http.MethodGet, "/v1/movies/:id", app.requirePermission("movies:read", app.showMovieHandler))
    router.HandleFunc(http.MethodPatch, "/v1/movies/:id", app.requirePermission("movies:write", app.updateMovieHandler))
    router.HandleFunc(http.MethodDelete, "/v1/movies/:id", app.requirePermission("movies:write", app.deleteMovieHandler))

    router.HandleFunc(http.MethodPost, "/v1/users", app.registerUserHandler)
    router.HandleFunc(http.MethodPut, "/v1/users/activated", app.activateUserHandler)

    router.HandleFunc(http.MethodPost, "/v1/tokens/authentication", app.createAuthenticationTokenHandler)

    // Register a new GET /debug/vars endpoint pointing to the expvar handler.
    router.Handler(http.MethodGet, "/debug/vars", expvar.Handler())

    return app.recoverPanic(app.enableCORS(app.rateLimit(app.authenticate(router))))
}
```

Note: Using the endpoint `GET /debug/vars` for the `expvar` handler is *conventional* but certainly not necessary. If you prefer, it's perfectly fine to register it at an alternative endpoint like `GET /v1/metrics` instead.

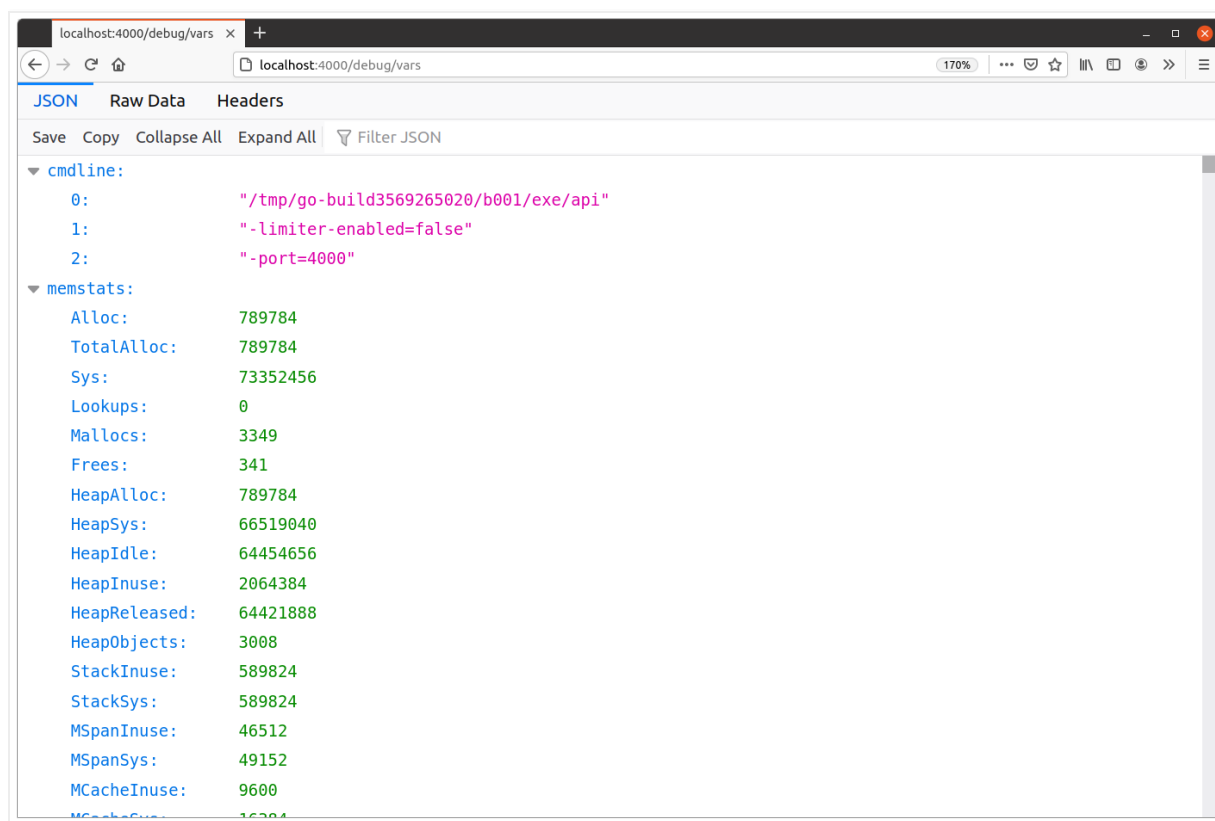
OK, let's try this out.

Go ahead and restart the API, passing in a couple of command line flags for demonstration purposes. Like so:

```
$ go run ./cmd/api -limiter-enabled=false -port=4000
{"level":"INFO","time":"2021-04-17T18:20:12Z","message":"database connection pool established"}
{"level":"INFO","time":"2021-04-17T18:20:12Z","message":"starting server","properties":{"addr":":4000","env":"development"}}
```

And if you visit <http://localhost:4000/debug/vars> in your web browser, you should see a JSON response containing information about your running application.

In my case, the response looks like this:



We can see that the JSON here currently contains two top-level items: `"cmdline"` and `"memstats"`. Let's quickly talk through what these represent.

The `"cmdline"` item contains an array of the command-line arguments used to run the application, beginning with the program name. This is essentially a JSON representation of the `os.Args` variable, and it's useful if you want to see exactly what non-default settings were used when starting the application.

The `"memstats"` item contains a 'moment-in-time' snapshot of memory usage, as returned by the `runtime.MemStats()` function. Documentation and descriptions for all of the values can be [found here](#), but the most important ones are:

- `TotalAlloc` — Cumulative bytes allocated on the heap (will not decrease).
- `HeapAlloc` — Current number of bytes on the heap.
- `HeapObjects` — Current number of objects on the heap.
- `Sys` — Total bytes of memory obtained from the OS (i.e. total memory reserved by the Go runtime for the heap, stacks, and other internal data structures).
- `NumGC` — Number of completed garbage collector cycles.
- `NextGC` — The target heap size of the next garbage collector cycle (Go aims to keep $\text{HeapAlloc} \leq \text{NextGC}$).

Hint: If any of the terms above are unfamiliar to you, then I strongly recommend reading the [Understanding Allocations in Go](#) blog post which provides a great introduction to how Go allocates memory and the concepts of the heap and the stack.

Creating Custom Metrics

The default information exposed by the `expvar` handler is a good start, but we can make it even more useful by exposing some additional custom metrics in the JSON response.

To illustrate this, we'll start really simple and first expose our application version number in the JSON. If you don't remember, the version number is currently defined as the string constant `"1.0.0"` in our `main.go` file.

The code to do this breaks down into two basic steps: first we need to register a custom variable with the `expvar` package, and then we need to set the value for the variable itself. In one line, the code looks roughly like this:

```
expvar.NewString("version").Set(version)
```

The first part of this — `expvar.NewString("version")` — creates a new `expvar.String` type, then *publishes it* so it appears in the `expvar` handler's JSON response with the name `"version"`, and then returns a pointer to it. Then we use the `Set()` method on it to assign an actual value to the pointer.

Two other things to note:

- The `expvar.String` type is safe for concurrent use. So — if you want to — it's OK to manipulate this value at runtime from your application handlers.
- If you try to register two `expvar` variables with the same name, you'll get a runtime panic when the duplicate variable is registered.

Let's go ahead and integrate this code into our `main()` function, like so:

File: cmd/api/main.go

```
package main

import (
    "context"
    "database/sql"
    "expvar" // New import
    "flag"
    "os"
    "strings"
    "sync"
    "time"

    "greenlight.alexedwards.net/internal/data"
    "greenlight.alexedwards.net/internal/jsonlog"
    "greenlight.alexedwards.net/internal/mailer"

    _ "github.com/lib/pq"
)

// Remember, our version number is just a constant string (for now).
const version = "1.0.0"

...

func main() {
    ...

    // Publish a new "version" variable in the expvar handler containing our application
    // version number (currently the constant "1.0.0").
    expvar.NewString("version").Set(version)

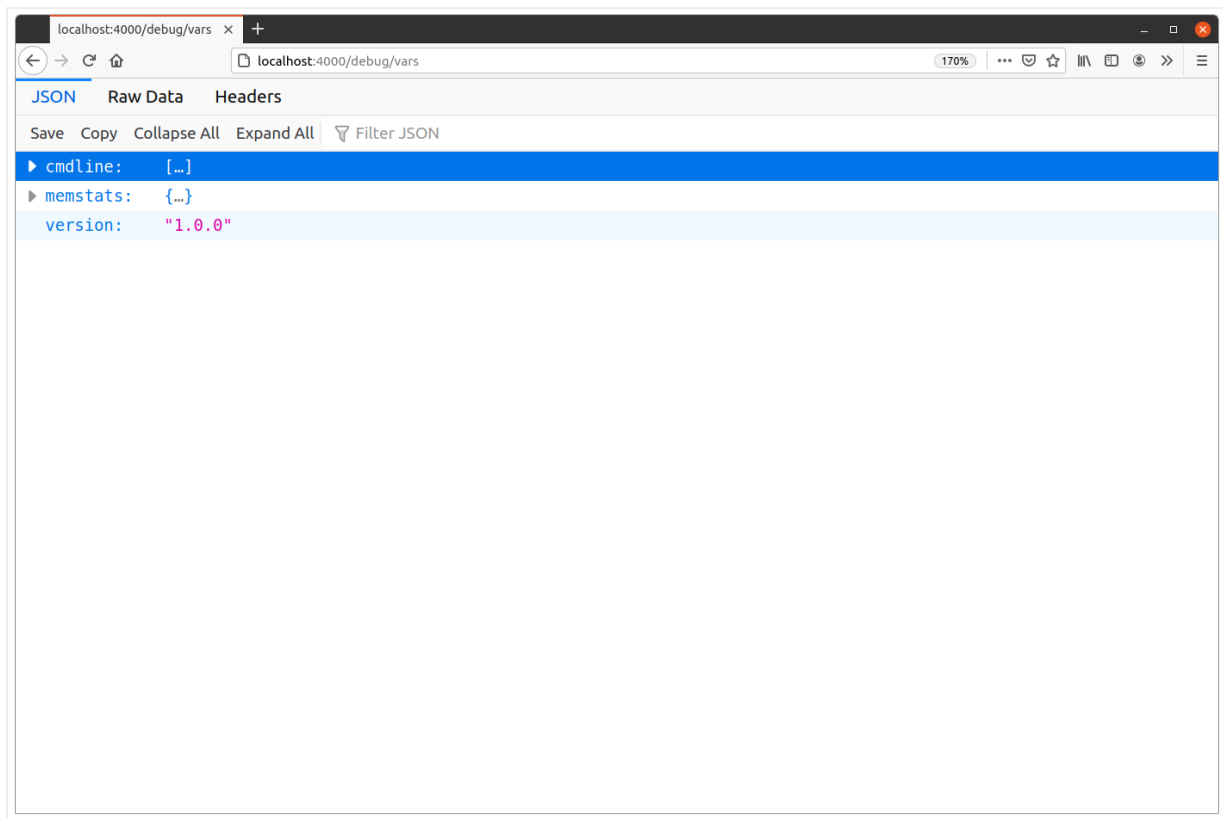
    app := &Application{
        config: cfg,
        logger: logger,
        models: data.NewModels(db),
        mailer: mailer.New(cfg.smtp.host, cfg.smtp.port, cfg.smtp.username, cfg.smtp.password, cfg.smtp.sender),
    }

    err = app.serve()
    if err != nil {
        logger.PrintFatal(err, nil)
    }
}

...
```

If you restart the API and visit <http://localhost:4000/debug/vars> in your web browser again, you should now see a **"version": "1.0.0"** item in the JSON.

Similar to this:



Note: In the code above we used the `expvar.NewString()` function to register and publish a string in the `expvar` handler. But Go also provides functions for a few other common data types: `NewFloat()`, `NewInt()` and `NewMap()`. All these work in a very similar way, and we'll put them to use in the next chapter.

Dynamic metrics

Occasionally you might want to publish metrics which require you to call other code — or do some kind of pre-processing — to generate the necessary information. To help with this there is the `expvar.Publish()` function, which allows you to publish the *result of a function* in the JSON output.

For example, if you want to publish the number of currently active goroutines from Go's `runtime.NumGoroutine()` function, you could write the following code:

```
expvar.Publish("goroutines", expvar.Func(func() interface{} {  
    return runtime.NumGoroutine()  
}))
```

It's important to point out here that the `interface{} value returned from this function must`

encode to JSON without any errors. If it can't be encoded to JSON, then it will be omitted from the `expvar` output and the response from the `GET /debug/vars` endpoint will be malformed. Any error will be silently discarded.

In the case of the code snippet above, `runtime.NumGoroutine()` returns a regular `int` type — which will encode to a JSON number. So there's no problem with that here.

OK, let's add this code to our `main()` function, along with two other functions which:

- Publish information about the state of our database connection pool (such as the number of idle and in-use connections) via the `db.Stats()` method.
- Publish the current Unix timestamp with second precision.

File: cmd/api/main.go

```
package main

import (
    "context"
    "database/sql"
    "expvar"
    "flag"
    "os"
    "runtime" // New import
    "strings"
    "sync"
    "time"

    "greenlight.alexedwards.net/internal/data"
    "greenlight.alexedwards.net/internal/jsonlog"
    "greenlight.alexedwards.net/internal/mailer"

    _ "github.com/lib/pq"
)

...

func main() {
    ...

    expvar.NewString("version").Set(version)

    // Publish the number of active goroutines.
    expvar.Publish("goroutines", expvar.Func(func() interface{} {
        return runtime.NumGoroutine()
    }))

    // Publish the database connection pool statistics.
    expvar.Publish("database", expvar.Func(func() interface{} {
        return db.Stats()
    }))

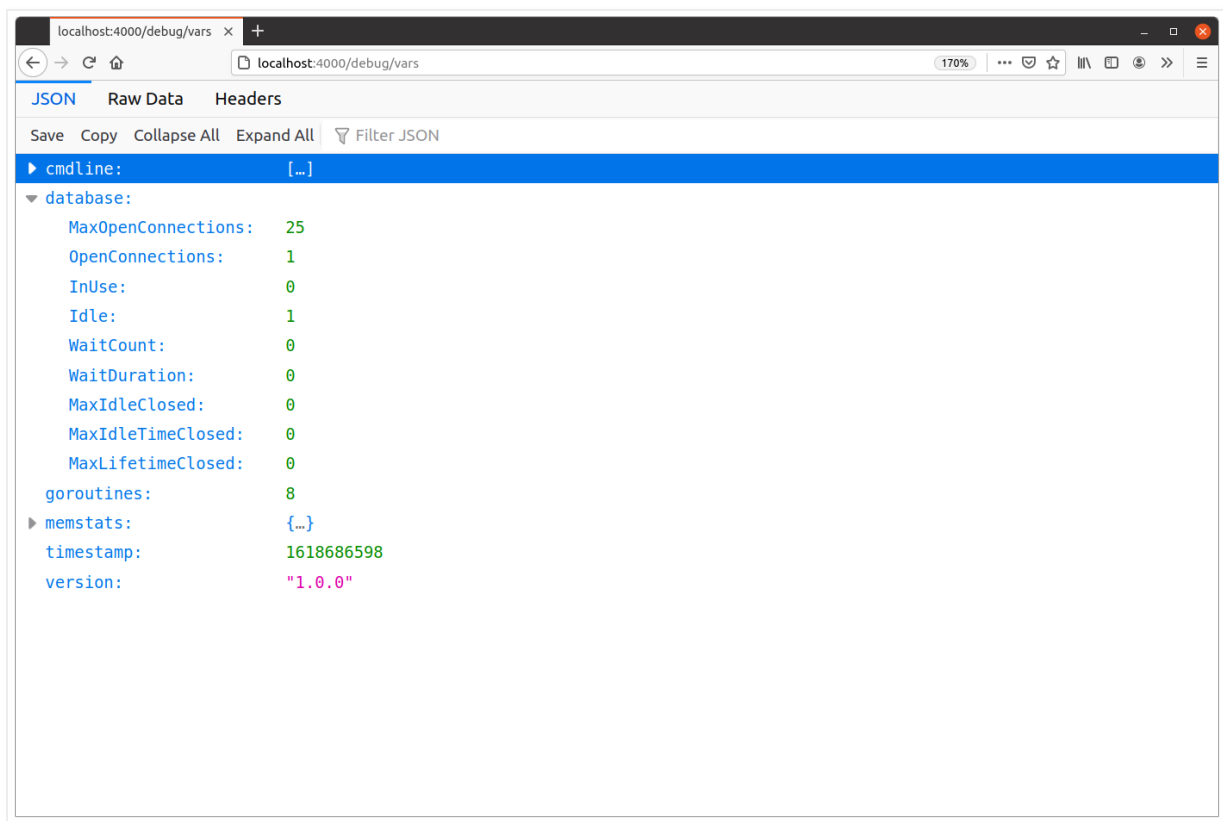
    // Publish the current Unix timestamp.
    expvar.Publish("timestamp", expvar.Func(func() interface{} {
        return time.Now().Unix()
    }))

    app := &application{
        config: cfg,
        logger: logger,
        models: data.NewModels(db),
        mailer: mailer.New(cfg.smtp.host, cfg.smtp.port, cfg.smtp.username, cfg.smtp.password, cfg.smtp.sender),
    }

    err = app.serve()
    if err != nil {
        logger.PrintFatal(err, nil)
    }
}

...
```

If you restart the API and open the `GET /debug/vars` endpoint in your browser again, you should now see the additional `"database"`, `"goroutines"` and `"timestamp"` items in the JSON. Like so:



In my case, I can see that the application currently has 8 active goroutines, and the database connection pool is in its 'initial' state with just one currently idle connection (which was created when our code called `db.PingContext()` on startup).

If you like, you can use a tool like [hey](#) to generate some requests to your application and see how these figures change under load. For example, you can send a batch of requests to the `POST /v1/tokens/authentication` endpoint (which is slow and costly because it checks a bcrypt-hashed password) like so:

Summary:

```
Total data: 24975 bytes
Size/request: 124 bytes
```

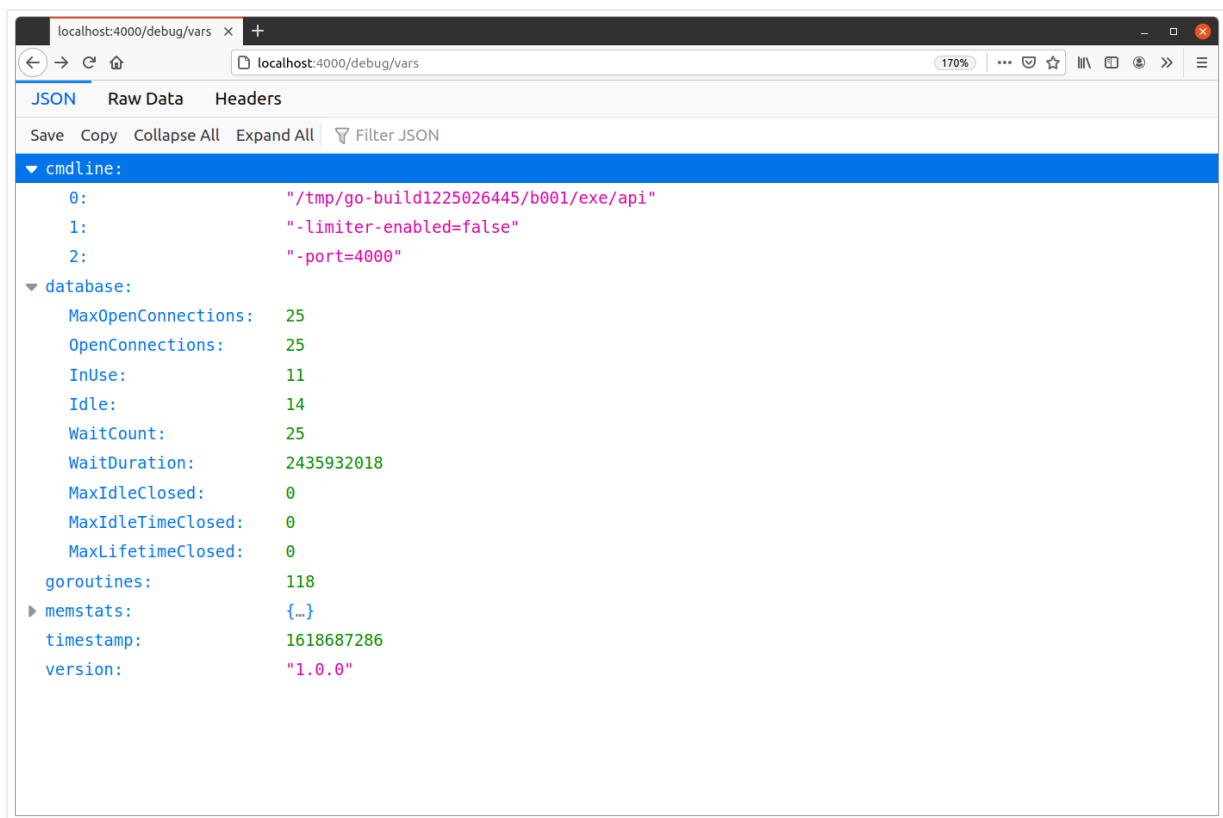
[illegible]

```
10% in 1.8143 secs
25% in 1.8871 secs
50% in 1.9867 secs
75% in 2.1000 secs
90% in 2.2017 secs
95% in 2.2642 secs
99% in 2.3799 secs
```

```
DNS+dialup: 0.0009 secs, 1.6169 secs, 2.4612 secs
DNS-lookup: 0.0005 secs, 0.0000 secs, 0.0030 secs
req write: 0.0002 secs, 0.0000 secs, 0.0051 secs
resp wait: 1.9924 secs, 1.6168 secs, 2.4583 secs
resp read: 0.0000 secs, 0.0000 secs, 0.0001 secs
```

[201] 200 responses

If you visit the `GET /debug/vars` endpoint while the `hey` tool is running, you should see that your application metrics now look quite different:



At the moment I took this screenshot, we can see that my API application had 118 active goroutines, with 11 database connections in use and 14 connections sat idle.

There are a couple of other interesting things to point out too.

The database **WaitCount** figure of 25 is the total number of times that our application had to wait for a database connection to become available in our **sql.DB** pool (because all connections were in-use). Likewise, **WaitCountDuration** is the cumulative amount of time (in nanoseconds) spent waiting for a connection. From these, it's possible to calculate that *when our application did have to wait for a database connection, the average wait time was approximately 98 milliseconds*. Ideally, you want to be seeing zeroes or very low numbers for these two things in production.

Also, the **MaxIdleTimeClosed** figure is the total count of the number of connections that have been closed because they reached their **ConnMaxIdleTime** limit (which in our case is set to 15 minutes by default). If you leave the application running but don't use it, and come back in 15 minutes time, you should see that the number of open connections has dropped to zero and the **MaxIdleTimeClosed** count has increased accordingly.

You might like to play around with this and try changing some of the configuration parameters for the connection pool to see how it affects the behavior of these figures under load. For example:

```
$ go run ./cmd/api -limiter-enabled=false -db-max-open-conns=50 -db-max-idle-conns=50 -db-max-idle-time=20s -port=4000
```

Additional Information

Protecting the metrics endpoint

It's important to be aware that these metrics provide very useful information to anyone who wants to perform a denial-of-service attack against your application, and that the `"cmdline"` values may also expose potentially sensitive information (like a database DSN).

So you should make sure to restrict access to the `GET /debug/vars` endpoint when running in a production environment.

There are a few different approaches you could take to do this.

One option is to leverage our existing authentication process and create a `metrics:view` permission so that only certain trusted users can access the endpoint. Another option would be to use HTTP Basic Authentication to restrict access to the endpoint.

In our case, when we deploy our application in production later we will run it behind [Caddy](#) as a reverse proxy. As part of our Caddy set up, we'll restrict access to the `GET /debug/vars` endpoint so that it can only be accessed via connections from the local machine, rather than being exposed on the internet.

Removing default metrics

It's currently not possible to remove the default `"cmdline"` and `"memstats"` items from the `expvar` handler, even if you want to. There's an [open issue](#) regarding this, and hopefully it will become possible to omit these in a future version of Go.

Request-level Metrics

In this chapter we're going to create some new middleware to record custom *request-level* metrics for our application. We'll start by recording the following three things:

- The total number of requests received.
- The total number of responses sent.
- The total (cumulative) time taken to process all requests in [microseconds](#).

All these values will be integers, so we'll be able to register these metrics with the `expvar` package using the `expvar.NewInt()` function.

Let's jump straight into the code and create a new `metrics()` middleware method, which initializes the necessary `expvar` variables and then updates them each time we process a request. Like so:

File: cmd/api/middleware.go

```
package main

import (
    "errors"
    "expvar" // New import
    "fmt"
    "net"
    "net/http"
    "strings"
    "sync"
    "time"

    "greenlight.alexedwards.net/internal/data"
    "greenlight.alexedwards.net/internal/validator"

    "golang.org/x/time/rate"
)

...

func (app *application) metrics(next http.Handler) http.Handler {
    // Initialize the new expvar variables when the middleware chain is first built.
    totalRequestsReceived := expvar.NewInt("total_requests_received")
    totalResponsesSent := expvar.NewInt("total_responses_sent")
    totalProcessingTimeMicroseconds := expvar.NewInt("total_processing_time_us")

    // The following code will be run for every request...
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        // Record the time that we started to process the request.
        start := time.Now()

        // Use the Add() method to increment the number of requests received by 1.
        totalRequestsReceived.Add(1)

        // Call the next handler in the chain.
        next.ServeHTTP(w, r)

        // On the way back up the middleware chain, increment the number of responses
        // sent by 1.
        totalResponsesSent.Add(1)

        // Calculate the number of microseconds since we began to process the request,
        // then increment the total processing time by this amount.
        duration := time.Since(start).Microseconds()
        totalProcessingTimeMicroseconds.Add(duration)
    })
}
```

Once that's done we need to update the `cmd/api/routes.go` file to include the new `metrics()` middleware, right at the beginning of the chain. Like so:

File: cmd/api/routes.go

```
package main

...

func (app *application) routes() http.Handler {
    router := httprouter.New()

    router.NotFound = http.HandlerFunc(app.notFoundResponse)
    router.MethodNotAllowed = http.HandlerFunc(app.methodNotAllowedResponse)

    router.HandleFunc(http.MethodGet, "/v1/healthcheck", app.healthcheckHandler)

    router.HandleFunc(http.MethodGet, "/v1/movies", app.requirePermission("movies:read", app.listMoviesHandler))
    router.HandleFunc(http.MethodPost, "/v1/movies", app.requirePermission("movies:write", app.createMovieHandler))
    router.HandleFunc(http.MethodGet, "/v1/movies/:id", app.requirePermission("movies:read", app.showMovieHandler))
    router.HandleFunc(http.MethodPatch, "/v1/movies/:id", app.requirePermission("movies:write", app.updateMovieHandler))
    router.HandleFunc(http.MethodDelete, "/v1/movies/:id", app.requirePermission("movies:write", app.deleteMovieHandler))

    router.HandleFunc(http.MethodPost, "/v1/users", app.registerUserHandler)
    router.HandleFunc(http.MethodPut, "/v1/users/activated", app.activateUserHandler)

    router.HandleFunc(http.MethodPost, "/v1/tokens/authentication", app.createAuthenticationTokenHandler)

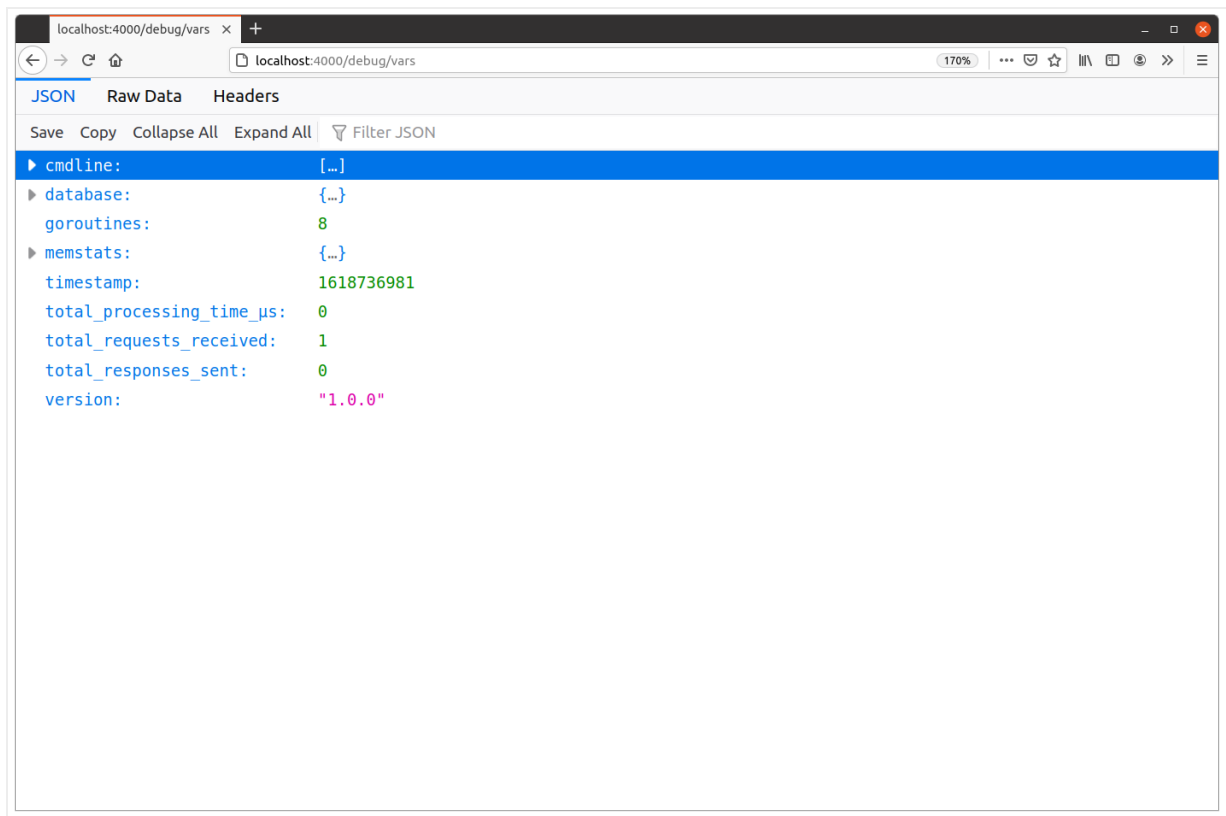
    router.Handler(http.MethodGet, "/debug/vars", expvar.Handler())

    // Use the new metrics() middleware at the start of the chain.
    return app.metrics(app.recoverPanic(app.enableCORS(app.rateLimit(app.authenticate(router)))))
}
```

OK, let's try this out. Go ahead and run the API with rate limiting disabled again:

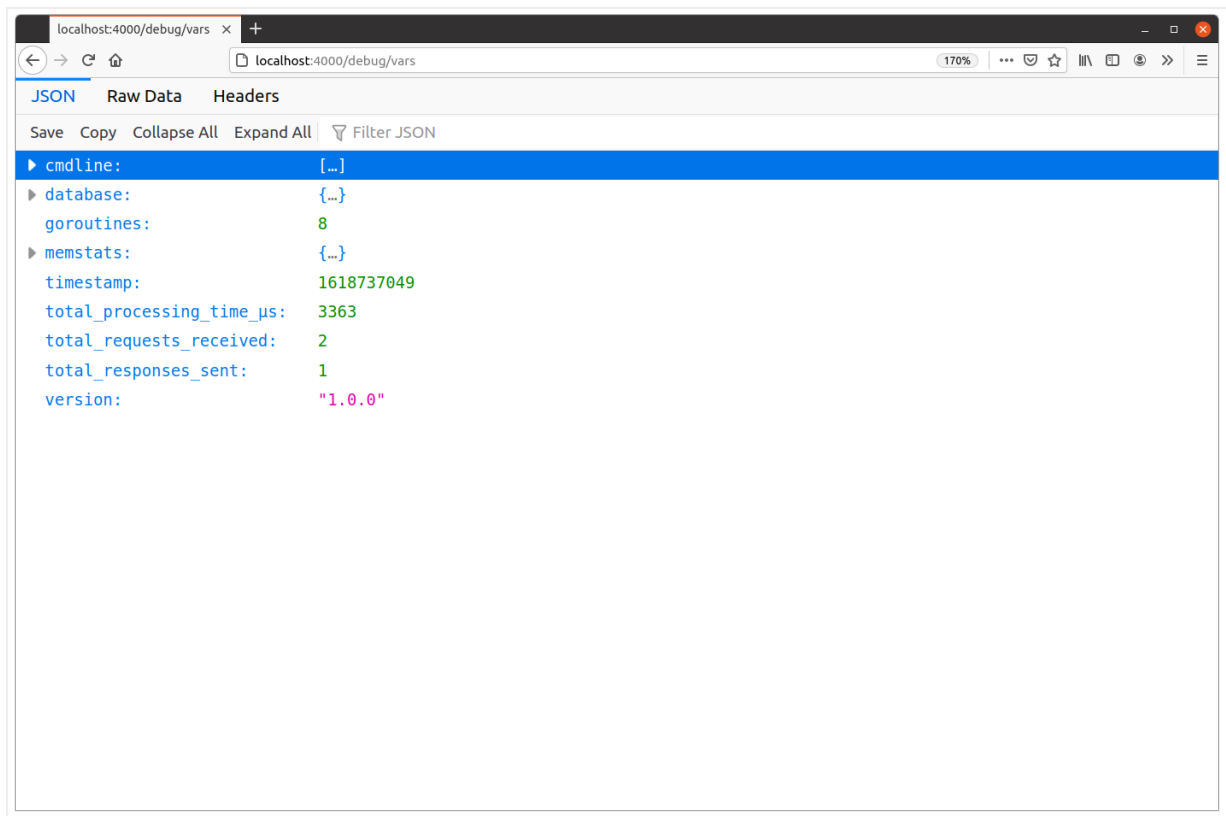
```
$ go run ./cmd/api -limiter-enabled=false
{"level":"INFO","time":"2021-04-18T09:06:05Z","message":"database connection pool established"}
{"level":"INFO","time":"2021-04-18T09:06:05Z","message":"starting server","properties":{"addr":":4000","env":"development"}}}
```

And when you visit localhost:4000/debug/vars in your browser, you should now see that the JSON response includes items for `total_requests_received`, `total_responses_sent` and `total_processing_time_µs`. Like so:



At this point, we can see that our API has received one request and sent zero responses. That makes sense if you think about it — at the moment that *this JSON response* was generated, it hadn't actually been sent.

If you refresh the page, you should see these numbers increment — including an increase in the `total_processing_time_us` value:



Let's try this out under load and use **hey** to make some requests to the **POST /v1/tokens/authentication** endpoint again.

First restart the API:

```
$ go run ./cmd/api -limiter-enabled=false
{"level":"INFO","time":"2021-04-18T09:18:36Z","message":"database connection pool established"}
{"level":"INFO","time":"2021-04-18T09:18:36Z","message":"starting server","properties":{"addr":":4000","env":"development"}}
```

And then use **hey** to generate the load like so:

Summary:

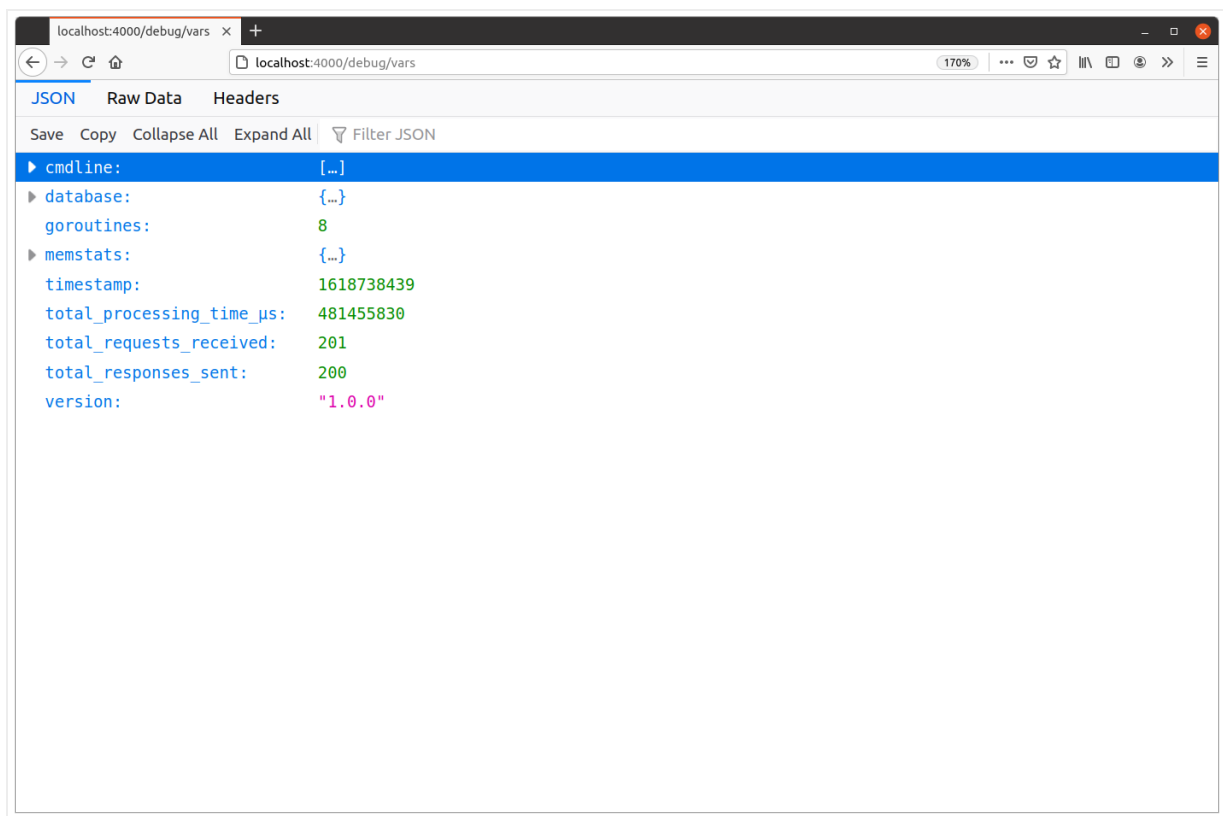
```
Total data: 24987 bytes
Size/request: 124 bytes
```

[illegible]

```
10% in 2.1386 secs
25% in 2.2678 secs
50% in 2.4197 secs
75% in 2.5769 secs
90% in 2.7718 secs
95% in 2.9125 secs
99% in 3.0220 secs
```

```
DNS+dialup: 0.0007 secs, 1.6788 secs, 3.0333 secs
DNS-lookup: 0.0005 secs, 0.0000 secs, 0.0047 secs
req write: 0.0001 secs, 0.0000 secs, 0.0012 secs
resp wait: 2.4293 secs, 1.6787 secs, 3.0293 secs
resp read: 0.0000 secs, 0.0000 secs, 0.0001 secs
```

[201] 200 responses



Based on the `total_processing_time_us` and `total_responses_sent` values, we can calculate that the average processing time per request was approximately 2.4 seconds (remember, this endpoint is deliberately slow and computationally expensive since we need to verify the bcrypted user password).

```
481455830 µs / 201 reqs = 2395303 µs/req = 2.4 s/req
```

This aligns with the data that we're seeing in the results summary from `hey`, which gives the average response time as 2.4302 secs. The `hey` figure is the time for the complete round-trip, so it will always be a little higher as it includes network latency and the time taken for `http.Server` to manage the request.

Additional Information

Calculating additional metrics

Based on this information in the `GET /debug/vars` response, you can also derive some additional interesting metrics. Such as...

- The number of 'active' in-flight requests:

```
total_requests_received - total_responses_sent
```

- The average number of requests received per second (between calls A and B to the `GET /debug/vars` endpoint):

```
(total_requests_received_B - total_requests_received_A) / (timestamp_B - timestamp_A)
```

- The average processing time per request (between calls A and B to the `GET /debug/vars` endpoint):

```
(total_processing_time_μs_B - total_processing_time_μs_A) / (total_requests_received_B - total_requests_received_A)
```

Recording HTTP Status Codes

As well as recording the total count of responses sent, we can take this further and extend our `metrics()` middleware to start tallying exactly which HTTP status codes our responses had.

The tricky part of doing this is finding out *what HTTP status code a response has* in our `metrics()` middleware. Unfortunately Go doesn't make this easy — there is no built-in way to examine a `http.ResponseWriter` to see what status code is going to be sent to a client.

The de-facto workaround is to create your own [custom implementation](#) of `http.ResponseWriter` which records a copy of the HTTP status code for future access. But doing this can be quite brittle and awkward — there are several edge cases that you need to be wary of, and it can cause problems if you are using any of the 'additional' `ResponseWriter` interfaces such as `http.Flusher` and `http.Hijacker`.

Rather than making your own custom `http.ResponseWriter` implementation, I highly recommend using the third-party [httpsnoop](#) package. It's small and focused, with no additional dependencies, and it makes it very easy to record the HTTP status code and size of each response, along with the total processing time for each request.

If you're following along, go ahead and download it like so:

```
$ go get github.com/felixge/httpsnoop@v1.0.2
go: downloading github.com/felixge/httpsnoop v1.0.2
go get: added github.com/felixge/httpsnoop v1.0.2
```

The key feature of `httpsnoop` that we're going to use is the `httpsnoop.CaptureMetrics()` function, which looks like this:

```
func CaptureMetrics(hnd http.Handler, w http.ResponseWriter, r *http.Request) Metrics
```

This function essentially wraps a `http.Handler`, executes the handler, and then returns a `Metrics` struct containing the following information:

```

type Metrics struct {
    // Code is the first http response status code passed to the WriteHeader() method of
    // the ResponseWriter. If no such call is made, a default code of 200 is
    // assumed instead.
    Code int
    // Duration is the time it took to execute the handler.
    Duration time.Duration
    // Written is the number of bytes successfully written by the Write() method of the
    // ResponseWriter. Note that ResponseWriters may also write data to their underlying
    // connection directly, but those writes are not tracked.
    Written int64
}

```

Let's dig in and adapt our `metrics()` middleware to use the `httpsnoop` package, and record the HTTP status codes for our responses.

To do this, we're going to publish a new `total_responses_sent_by_status` variable using the `expvar.NewMap()` function. This will give us a map in which we can store the different HTTP status codes, along with a running count of responses for each status. Then, for each request, we'll use the `httpsnoop.CaptureMetrics()` function to get the HTTP status code, and increment the value in the map accordingly.

While we're at it, let's also remove the code that we made earlier for recording the total processing time, and leverage the functionality built in to `httpsnoop` instead.

Go ahead and update the `metrics()` middleware as follows:

File: cmd/api/middleware.go

```
package main

import (
    "errors"
    "expvar"
    "fmt"
    "net"
    "net/http"
    "strconv" // New import
    "strings"
    "sync"
    "time"

    "greenlight.alexedwards.net/internal/data"
    "greenlight.alexedwards.net/internal/validator"

    "github.com/felixge/httpsnoop" // New import
    "golang.org/x/time/rate"
)

...

func (app *application) metrics(next http.Handler) http.Handler {
    totalRequestsReceived := expvar.NewInt("total_requests_received")
    totalResponsesSent := expvar.NewInt("total_responses_sent")
    totalProcessingTimeMicroseconds := expvar.NewInt("total_processing_time_us")
    // Declare a new expvar map to hold the count of responses for each HTTP status
    // code.
    totalResponsesSentByStatus := expvar.NewMap("total_responses_sent_by_status")

    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        // Increment the requests received count, like before.
        totalRequestsReceived.Add(1)

        // Call the httpsnoop.CaptureMetrics() function, passing in the next handler in
        // the chain along with the existing http.ResponseWriter and http.Request. This
        // returns the metrics struct that we saw above.
        metrics := httpsnoop.CaptureMetrics(next, w, r)

        // Increment the response sent count, like before.
        totalResponsesSent.Add(1)

        // Get the request processing time in microseconds from httpsnoop and increment
        // the cumulative processing time.
        totalProcessingTimeMicroseconds.Add(metrics.Duration.Microseconds())

        // Use the Add() method to increment the count for the given status code by 1.
        // Note that the expvar map is string-keyed, so we need to use the strconv.Itoa()
        // function to convert the status code (which is an integer) to a string.
        totalResponsesSentByStatus.Add(strconv.Itoa(metrics.Code), 1)
    })
}
```

Alright, let's try this out. Run the API again, but this time *leave the rate limiter enabled*. Like so:

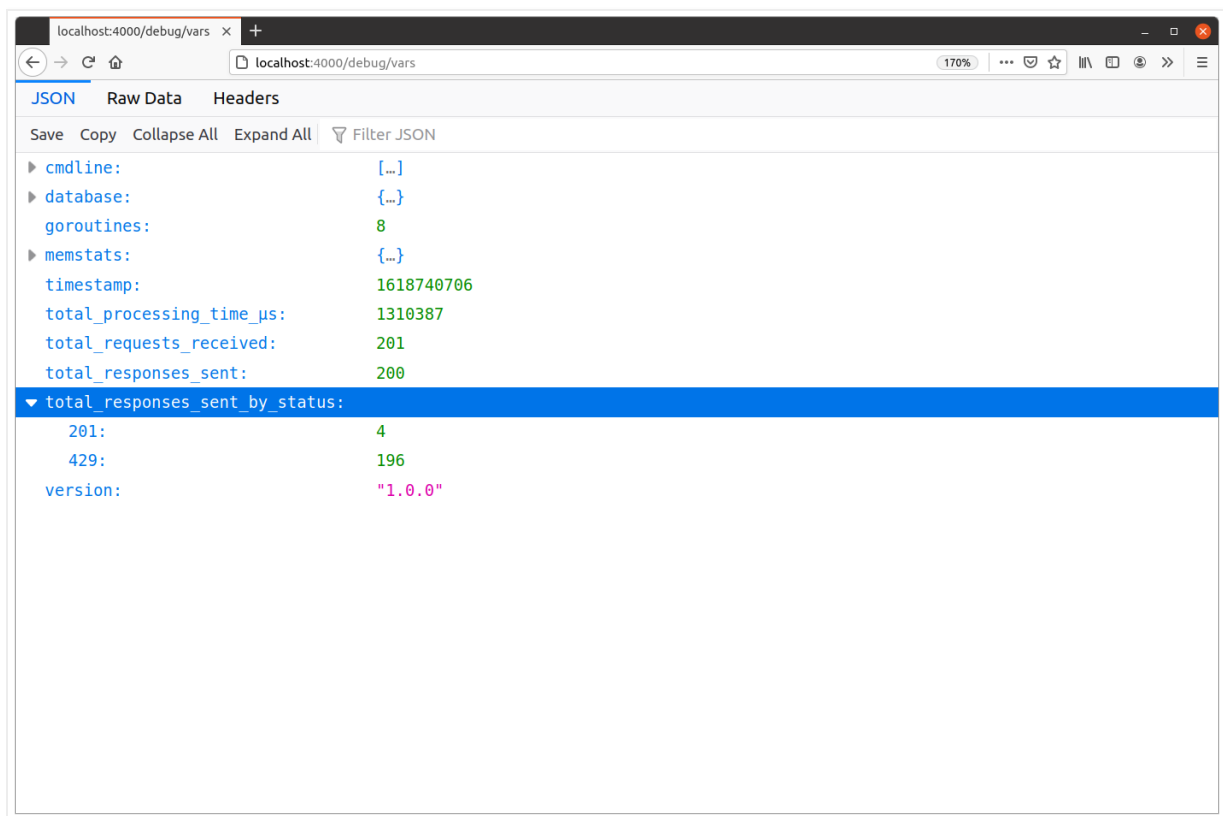

```
$ go run ./cmd/api
{"level":"INFO","time":"2021-04-18T10:09:09Z","message":"database connection pool established"}
{"level":"INFO","time":"2021-04-18T10:09:09Z","message":"starting server","properties":{"addr":":4000","env":"development"}}
```

Then use **hey** again to generate some load against the **POST /v1/tokens/authentication** endpoint. This should result in a small number of successful **201 Created** responses, but many more **429 Too Many Requests** responses due to the rate limit being hit.

[illegible]

And if you take a look at the metrics in your browser, you should now see the same corresponding counts for each status code under the `total_responses_sent_by_status`

item, similar to this:



Additional Information

Visualizing and analyzing metrics

Now that we have some good application-level metrics being recorded, there is the whole question of *what should you do with them?*

The answer to this will be different from project-to-project.

For some low-value applications, it might be sufficient to manually spot check the metrics every so often — or only when you suspect a problem — and make sure that nothing looks unusual or out of place.

In other projects, you might want to write a script to periodically fetch the JSON data from the `GET /debug/vars` endpoint and carry out further analysis. This might include functionality to alert you if something appears to be abnormal.

At the other end of the spectrum, you might want to use a tool like [Prometheus](#) to fetch and

visualize the data from the endpoint, and display graphs of the metrics in real-time.

There are a lot of different options, and the right thing to do really depends on the needs of your project and business. But in all cases, using the `expvar` package to collect and publish the metrics gives you a great platform from which you can integrate any external monitoring, alerting or visualization tools.

Building, Versioning and Quality Control

In this section of the book we're going to shift our focus from writing code to *managing and maintaining* our project, and take steps to help automate common tasks and prepare our API for deployment.

Specifically, you'll learn how to:

- Use a makefile to automate common tasks in your project, such as creating and executing migrations.
- Carry out quality control checks of your code using the `go vet` and `staticcheck` tools.
- Vendor third-party packages, in case they ever become unavailable in the future.
- Build and run executable binaries for your applications, reduce their size, and cross-compile binaries for different platforms.
- Burn-in a version number and build time to your application when building the binary.
- Leverage Git to generate automated version numbers as part of your build process.

Creating and Using Makefiles

In this first chapter we're going to look at how to use the GNU `make` utility and makefiles to help automate common tasks in your project.

The `make` tool should be pre-installed in most Linux distributions, but if it's not already on your machine you should be able to install it via your package manager. For example, if your OS supports the `apt` package manager (like Debian and Ubuntu does) you can install it with:

```
$ sudo apt install make
```

It's also probably already on your machine if you use macOS, but if not you can use `brew` to install it:

```
$ brew install make
```

On Windows machines you can install `make` using the `Chocolatey package manager` with the command:

```
> choco install make
```

A simple makefile

Now that the `make` utility is installed on your system, let's create our first iteration of a *makefile*. We'll start simple, and then build things up step-by-step.

A makefile is essentially a text file which contains one or more *rules* that the `make` utility can run. Each rule has a *target* and contains a sequence of sequential *commands* which are executed when the rule is run. Generally speaking, makefile rules have the following structure:

```
# comment (optional)
target:
  command
  command
  ...
```

Important: Please note that each command in a makefile rule must start with a tab character, **not spaces**. If you happen to be reading this in PDF format, the tab will appear as a single space character in the snippet above, but please make sure it is a tab character in your own code.

If you've been following along, you should already have an empty **Makefile** in the root of your project directory. Let's jump in and create a rule which executes the **go run ./cmd/api** command to run our API application. Like so:

```
File: Makefile
```

```
run:
  go run ./cmd/api
```

Make sure that the **Makefile** is saved, and then you can execute a specific rule by running **\$ make <target>** from your terminal.

Let's go ahead and call **make run** to start the API:

```
$ make run
go run ./cmd/api
{"level":"INFO","time":"2021-04-18T10:38:26Z","message":"database connection pool established"}
{"level":"INFO","time":"2021-04-18T10:38:26Z","message":"starting server","properties":{"addr":":4000","env":"development"}}
```

Great, that's worked well. When we type **make run**, the make utility looks for a file called **Makefile** or **makefile** in the current directory and then executes the commands associated with the **run** target.

One thing to point out — by default **make** echoes commands in the terminal output. We can see that in the code above where the first line in the output is the echoed command **go run ./cmd/api**. If you want, it's possible to suppress commands from being echoed by prefixing them with the **@** character.

Environment variables

When we execute a **make** rule, every environment variable that is available to **make** when it starts is transformed into a *make variable* with the same name and value. We can then access these variables using the syntax **\${VARIABLE_NAME}** in our makefile.

To illustrate this, let's create two additional rules — a **psql** rule for connecting to our

database and an **up** rule to execute our database migrations. If you've been following along, both of these rules will need access to the database DSN value from your **GREENLIGHT_DB_DSN** environment variable.

Go ahead and update your **Makefile** to include these two new rules like so:

File: Makefile

```
run:
  go run ./cmd/api

psql:
  psql ${GREENLIGHT_DB_DSN}

up:
  @echo 'Running up migrations...'
  migrate -path ./migrations -database ${GREENLIGHT_DB_DSN} up
```

Notice here how we've used the **@** character in the **up** rule to prevent the **echo** command *from being echoed itself* when running?

OK, let's try this out by running **make up** to execute our database migrations:

```
$ make up
Running up migrations...
migrate -path ./migrations -database postgres://greenlight:pa55word@localhost/greenlight up
no change
```

You should see from the output here that the value of your **GREENLIGHT_DB_DSN** environment variable is successfully pulled through and used in the **make** rule. If you've been following along, there shouldn't be any outstanding migrations to apply, so this should then exit successfully with no further action.

We're also starting to see the benefits of using a makefile here — being able to type **make up** is a big improvement on having to remember and use the full command for executing our 'up' migrations.

Likewise, if you want, you can also try running **make psql** to connect to the **greenlight** database with **psql**.

Passing arguments

The **make** utility also allows you to pass *named arguments* when executing a particular rule. To illustrate this, let's add a **migration** rule to our makefile to generate a new pair of

migration files. The idea is that when we execute this rule we'll pass the name of the migration files as an argument, similar to this:

```
$ make migration name=create_example_table
```

The syntax to access the value of named arguments is exactly the same as for accessing environment variables. So, in the example above, we could access the migration file name via `${name}` in our makefile.

Go ahead and update the makefile to include this new `migration` rule, like so:

File: Makefile

```
run:
  go run ./cmd/api

psql:
  psql ${GREENLIGHT_DB_DSN}

migration:
  @echo 'Creating migration files for ${name}...'
  migrate create -seq -ext=.sql -dir=./migrations ${name}

up:
  @echo 'Running up migrations...'
  migrate -path ./migrations -database ${GREENLIGHT_DB_DSN} up
```

And if you execute this new rule with the `name=create_example_table` argument you should see the following output:

```
$ make migration name=create_example_table
Creating migration files for create_example_table ...
migrate create -seq -ext=.sql -dir=./migrations create_example_table
/home/alex/Projects/greenlight/migrations/000007_create_example_table.up.sql
/home/alex/Projects/greenlight/migrations/000007_create_example_table.down.sql
```

You'll now also have two new empty migration files with the name `create_example_table` in your migrations folder. Like so:

```
$ ls ./migrations/
000001_create_movies_table.down.sql    000004_create_users_table.up.sql
000001_create_movies_table.up.sql      000005_create_tokens_table.down.sql
000002_add_movies_check_constraints.down.sql  000005_create_tokens_table.up.sql
000002_add_movies_check_constraints.up.sql    000006_add_permissions.down.sql
000003_add_movies_indexes.down.sql          000006_add_permissions.up.sql
000003_add_movies_indexes.up.sql           000007_create_example_table.down.sql
000004_create_users_table.down.sql         000007_create_example_table.up.sql
```


If you're following along, we won't actually be using these two new migration files, so feel free to delete them:

```
$ rm migrations/000007*
```

Note: Variable names in makefiles are case-sensitive, so `foo`, `F00`, and `Foo` all refer to different variables. The [make documentation](#) suggests using lower case letters for variable names that only serve an internal purpose in the makefile, and using upper case variable names otherwise.

Namespacing targets

As your makefile continues to grow, you might want to start *namespacing* your target names to provide some differentiation between rules and help organize the file. For example, in a large makefile rather than having the target name `up` it would be clearer to give it the name `db/migrations/up` instead.

I recommend using the `/` character as a namespace separator, rather a period, hyphen or the `:` character. In fact, the `:` character should be strictly avoided in target names as it can cause problems when using *target prerequisites* (something that we'll cover in a moment).

Let's update our target names to use some sensible namespaces, like so:

File: Makefile

```
run/api:
  go run ./cmd/api

db/psql:
  psql ${GREENLIGHT_DB_DSN}

db/migrations/new:
  @echo 'Creating migration files for ${name}...'
  migrate create -seq -ext=.sql -dir=./migrations ${name}

db/migrations/up:
  @echo 'Running up migrations...'
  migrate -path ./migrations -database ${GREENLIGHT_DB_DSN} up
```

And you should be able to execute the rules by typing the full target name when running `make`. For example:

```
$ make run/api
go run ./cmd/api
{"level":"INFO","time":"2021-04-18T10:50:23Z","message":"database connection pool established"}
{"level":"INFO","time":"2021-04-18T10:50:23Z","message":"starting server","properties":{"addr":":4000","env":"development"}}
```

A nice feature of using the `/` character as the namespace separator is that you get tab completion in the terminal when typing target names. For example, if you type `make db/migrations/` and then hit tab on your keyboard the remaining targets under the namespace will be listed. Like so:

```
$ make db/migrations/
new  up
```

Prerequisite targets and asking for confirmation

The general syntax for a makefile rule that I gave at start of this chapter was a slight simplification, because it's also possible to specify *prerequisite* targets.

```
target: prerequisite-target-1 prerequisite-target-2 ...
command
command
...
```

When you specify a prerequisite target for a rule, the corresponding commands for the prerequisite targets will be run *before* executing the actual target commands.

Let's leverage this functionality to ask the user for *confirmation to continue* before executing our `db/migrations/up` rule.

To do this, we'll create a new `confirm` target which asks the user `Are you sure? [y/N]` and exits with an error if they do not enter `y`. Then we'll use this new `confirm` target as a prerequisite for `db/migrations/up`.

Go ahead and update your `Makefile` as follows:

File: Makefile

```
# Create the new confirm target.
confirm:
@echo -n 'Are you sure? [y/N] ' && read ans && [ ${ans:-N} = y ]

run/api:
go run ./cmd/api

db/psql:
psql ${GREENLIGHT_DB_DSN}

db/migrations/new:
@echo 'Creating migration files for ${name}...'
migrate create -seq -ext=.sql -dir=./migrations ${name}

# Include it as prerequisite.
db/migrations/up: confirm
@echo 'Running up migrations...'
migrate -path ./migrations -database ${GREENLIGHT_DB_DSN} up
```

The code in the `confirm` target is taken from [this StackOverflow post](#). Essentially, what happens here is that we ask the user `Are you sure? [y/N]` and then read the response. We then use the code `[ ${ans:-N} = y ]` to evaluate the response — this will return `true` if the user enters `y` and `false` if they enter anything else. If a command in a makefile returns `false`, then `make` will stop running the rule and exit with an error message — essentially stopping the rule in its tracks.

Also, importantly, notice that we have set `confirm` as a prerequisite for the `db/migrations/up` target?

Let's try this out and see what happens when we enter `y`:

```
$ make db/migrations/up
Are you sure? [y/N] y
Running up migrations...
migrate -path ./migrations -database postgres://greenlight:pa55word@localhost/greenlight up
no change
```

That looks good — the commands in our `db/migrations/up` rule have been executed as we would expect.

In contrast, let's try the same thing again but enter any other letter when asked for confirmation. This time, `make` should exit *without* executing anything in the `db/migrations/up` rule. Like so:

```
$ make db/migrations/up
Are you sure? [y/N] n
make: *** [Makefile:3: confirm] Error 1
```

Using a `confirm` rule as a prerequisite target like this is a really nice re-usable pattern. Any time you have a makefile rule which does something destructive or dangerous, you can now just include `confirm` as a prerequisite target to ask the user for confirmation to continue.

Displaying help information

Another small thing that we can do to make our makefile more user-friendly is to include some comments and help functionality. Specifically, we'll prefix each rule in our makefile with a comment in the following format:

```
## <example target call>: <help text>
```

Then we'll create a new `help` rule which parses the makefile itself, extracts the help text from the comments using `sed`, formats them into a table and then displays them to the user.

If you're following along, go ahead and update your `Makefile` so that it looks like this:

File: Makefile

```
## help: print this help message
help:
  @echo 'Usage:'
  @sed -n 's/^##//p' ${MAKEFILE_LIST} | column -t -s ':' | sed -e 's/^/ /'

confirm:
  @echo -n 'Are you sure? [y/N] ' && read ans && [ ${ans:-N} = y ]

## run/api: run the cmd/api application
run/api:
  go run ./cmd/api

## db/psql: connect to the database using psql
db/psql:
  psql ${GREENLIGHT_DB_DSN}

## db/migrations/new name=$1: create a new database migration
db/migrations/new:
  @echo 'Creating migration files for ${name}...'
  migrate create -seq -ext=.sql -dir=./migrations ${name}

## db/migrations/up: apply all up database migrations
db/migrations/up: confirm
  @echo 'Running up migrations...'
  migrate -path ./migrations -database ${GREENLIGHT_DB_DSN} up
```

Note: `MAKEFILE_LIST` is a [special variable](#) which contains the name of the makefile being parsed by `make`.

And if you now execute the `help` target, you should get a response which lists all the available targets and the corresponding help text. Similar to this:

```
$ make help
Usage:
  help                print this help message
  run/api             run the cmd/api application
  db/psql             connect to the database using psql
  db/migrations/new name=$1 create a new database migration
  db/migrations/up    apply all up database migrations
```

I should also point out that positioning the `help` rule as the first thing in the `Makefile` is a deliberate move. If you run `make` *without specifying a target* then it will default to executing the first rule in the file.

So this means that if you try to run `make` without a target you'll now be presented with the help information, like so:

```
$ make
Usage:
  help                print this help message
  run/api             run the cmd/api application
  db/psql             connect to the database using psql
  db/migrations/new name=$1 create a new database migration
  db/migrations/up    apply all up database migrations
```

Phony targets

In this chapter we've been using `make` to execute *actions*, but another (and arguably, the primary) purpose of `make` is to help create files on disk where *the name of a target is the name of a file* being created by the rule.

If you're using `make` primarily to execute actions, like we are, then this can cause a problem if there is a file in your project directory *with the same path* as a target name.

If you want, you can demonstrate this problem by creating a file called `./run/api` in the root of your project directory, like so:

```
$ mkdir run && touch run/api
```

And then if you execute `make run/api`, instead of our API application starting up you'll get the following message:

```
$ make run/api
make: 'run/api' is up to date.
```

Because we already have a file on disk at `./run/api`, the `make` tool considers this rule to have already been executed and so returns the message that we see above without taking any further action.

To work around this, we can declare our makefile targets to be **phony targets**:

A phony target is one that is not really the name of a file; rather it is just a name for a rule to be executed.

To declare a target as phony, you can make it prerequisite of the special `.PHONY` target. The syntax looks like this:

```
.PHONY: target
target: prerequisite-target-1 prerequisite-target-2 ...
command
command
...
```

Let's go ahead and update our **Makefile** so that all our rules have phony targets, like so:

File: Makefile

```
## help: print this help message
.PHONY: help
help:
@echo 'Usage:'
@sed -n 's/^##//p' ${MAKEFILE_LIST} | column -t -s ':' | sed -e 's/^/ /'

.PHONY: confirm
confirm:
@echo -n 'Are you sure? [y/N] ' && read ans && [ $$ans:-N ] = y ]

## run/api: run the cmd/api application
.PHONY: run/api
run/api:
go run ./cmd/api

## db/psql: connect to the database using psql
.PHONY: db/psql
db/psql:
psql ${GREENLIGHT_DB_DSN}

## db/migrations/new name=$1: create a new database migration
.PHONY: db/migrations/new
db/migrations/new:
@echo 'Creating migration files for ${name}...'
migrate create -seq -ext=.sql -dir=./migrations ${name}

## db/migrations/up: apply all up database migrations
.PHONY: db/migrations/up
db/migrations/up: confirm
@echo 'Running up migrations...'
migrate -path ./migrations -database ${GREENLIGHT_DB_DSN} up
```

If you run `make run/api` again now, it should now correctly recognize this as a phony target and execute the rule for us:

```
$ make run/api
go run ./cmd/api -db-dsn=postgres://greenlight:pa55word@localhost/greenlight
{"level":"INFO","time":"2021-04-18T11:04:06Z","message":"database connection pool established"}
{"level":"INFO","time":"2021-04-18T11:04:06Z","message":"starting server","properties":{"addr":":4000","env":"development"}}
```

You might think that it's only necessary to declare targets as phony if you have a conflicting file name, but in practice *not declaring a target as phony when it actually is* can lead to bugs or confusing behavior. For example, imagine if in the future someone unknowingly creates a file called `confirm` in the root of the project directory. This would mean that our `confirm` rule is never executed, which in turn would lead to dangerous or destructive rules being executed without confirmation.

To avoid this kind of bug, if you have a makefile rule which carries out an action (rather than creating a file) then it's best to get into the habit of declaring it as phony.

If you're following along, you can go ahead and remove the contents of the `run` directory

that we just made. Like so:

```
$ rm -rf run/
```

All in all, this is shaping up nicely. Our makefile is starting to contain some helpful functionality, and we'll continue to add more to it over the next few chapters of this book.

Although this is one of the last things we're doing in this build, creating a makefile in the root of a project directory is normally one of the very *first* things I do when starting a project. I find that using a makefile for common tasks helps save both typing and mental overhead during development, and — in the longer term — it acts as a useful entry point and a reminder of how things work when you come back to a project after a long break.

Managing Environment Variables

Using the `make run/api` command to run our API application opens up an opportunity to tweak our command-line flags, and remove the default value for our database DSN from the `main.go` file. Like so:

File: cmd/api/main.go

```
package main

...

func main() {
    var cfg config

    flag.IntVar(&cfg.port, "port", 4000, "API server port")
    flag.StringVar(&cfg.env, "env", "development", "Environment (development|staging|production)")

    // Use the empty string "" as the default value for the db-dsn command-line flag,
    // rather than os.Getenv("GREENLIGHT_DB_DSN") like we were previously.
    flag.StringVar(&cfg.db.dsn, "db-dsn", "", "PostgreSQL DSN")

    ...
}
```

Instead, we can update our makefile so that the DSN value from the `GREENLIGHT_DB_DSN` environment variable is passed in as part of the rule. If you're following along, please go ahead and update the `run/api` rule as follows:

File: Makefile

```
...

## run/api: run the cmd/api application
.PHONY: run/api
run/api:
    go run ./cmd/api -db-dsn=${GREENLIGHT_DB_DSN}

...
```

This is a small change but a really nice one, because it means that the *default configuration values for our application no longer change depending on the operating environment*. The command-line flag values passed at runtime are the *sole* mechanism for configuring our application settings, and there are still no secrets hard-coded in our project files.

During development running our application remains nice and easy — all we need to do is

type `make run/api`, like so:

```
$ make run/api
go run ./cmd/api -db-dsn=postgres://greenlight:pa55word@localhost/greenlight
{"level":"INFO","time":"2021-04-18T11:17:49Z","message":"database connection pool established"}
{"level":"INFO","time":"2021-04-18T11:17:49Z","message":"starting server","properties":{"addr":":4000","env":"development"}}
```

Hint: If you're not comfortable with the DSN value (which contains a password) being displayed on your screen when you type `make run/api`, remember that you can use the `@` character in your makefile to suppress that command from being echoed.

Using a `.envrc` file

If you like, you could also *remove* the `GREENLIGHT_DB_DSN` environment variable from your `$HOME/.profile` or `$HOME/.bashrc` files, and store it in a `.envrc` file in the root of your project directory instead.

If you're following along, go ahead and create a new `.envrc` file like so:

```
$ touch .envrc
```

File: `.envrc`

```
export GREENLIGHT_DB_DSN=postgres://greenlight:pa55word@localhost/greenlight
```

You can then use a tool like `direnv` to automatically load the variables from the `.envrc` file into your current shell, or alternatively, you can add an `include` command at the top of your `Makefile` to load them instead. Like so:

File: `Makefile`

```
# Include variables from the .envrc file
include .envrc

## help: print this help message
.PHONY: help
help:
@echo 'Usage:'
@sed -n 's/^##//p' ${MAKEFILE_LIST} | column -t -s ':' | sed -e 's/^/ /'

...
```

This approach is particularly convenient in projects where you need to make frequent changes to your environment variables, because it means that you can just edit the `.envrc` file without needing to reboot your computer or run `source` after each change.

Another nice benefit of this approach is that it provides a degree of separation between variables if you're working on multiple projects on the same machine.

Important: If you use this approach and your `.envrc` file contains any secrets, you must be careful to not commit the file into a version control system (like Git or Mercurial).

In a few chapters time we'll start version controlling our codebase with Git, so let's preemptively add an ignore rule so that the `.envrc` file is never committed.

```
$ echo '.envrc' >> .gitignore
```

Quality Controlling Code

In this chapter we're going to focus on adding an `audit` rule to our `Makefile` to check, test and tidy up our codebase automatically. In particular, the rule will:

- Use the `go mod tidy` command to prune any unused dependencies from the `go.mod` and `go.sum` files, and add any missing dependencies.
- Use the `go mod verify` command to check that the dependencies on your computer (located in your module cache located at `$GOPATH/pkg/mod`) haven't been changed since they were downloaded and that they match the cryptographic hashes in your `go.sum` file. Running this helps ensure that the dependencies being used are the exact ones that you expect.
- Use the `go fmt ./...` command to format all `.go` files in the project directory, according to the Go standard. This will reformat files 'in place' and output the names of any changed files.
- Use the `go vet ./...` command to check all `.go` files in the project directory. The `go vet` tool runs a variety of *analyzers* which carry out static analysis of your code and warn you about things which might be wrong but won't be picked up by the compiler — such as unreachable code, unnecessary assignments, and badly-formed build tags.
- Use the `go test -race -vet=off ./...` command to run all tests in the project directory. By default, `go test` automatically executes a small subset of the `go vet` checks before running any tests, so to avoid duplication we'll use the `-vet=off` flag to turn this off. The `-race` flag enables Go's *race detector*, which can help pick up certain classes of race conditions while tests are running.
- Use the third-party `staticcheck` tool to carry out some *additional static analysis checks*.

If you're following along, you'll need to install the `staticcheck` tool on your machine at this point. The simplest way to do this is by running the `go install` command like so:

```
$ go install honnef.co/go/tools/cmd/staticcheck@latest
go: downloading honnef.co/go/tools v0.1.3
go: downloading golang.org/x/tools v0.1.0
go: downloading github.com/BurntSushi/toml v0.3.1
go: downloading golang.org/x/sys v0.0.0-20210119212857-b64e53b001e4
go: downloading golang.org/x/xerrors v0.0.0-20200804184101-5ec99f83aff1
go: downloading golang.org/x/mod v0.3.0
$ which staticcheck
/home/alex/go/bin/staticcheck
```

Note: If nothing is found by the `which` command, check that your `$GOPATH/bin` directory is on your system path. If you're not sure where your `$GOPATH` directory is, you can find out by running `go env GOPATH` from your terminal.

Once that's installed, let's go ahead and create a new `audit` rule in our makefile. While we're at it, let's also add some comment blocks to help organize and clarify the purpose of our different makefile rules, like so:

File: Makefile

```
include .envrc

# ===== #
# HELPERS
# ===== #

## help: print this help message
.PHONY: help
help:
    @echo 'Usage:'
    @sed -n 's/^##//p' ${MAKEFILE_LIST} | column -t -s ':' | sed -e 's/^/ /'

.PHONY: confirm
confirm:
    @echo -n 'Are you sure? [y/N] ' && read ans && [ ${ans:-N} = y ]

# ===== #
# DEVELOPMENT
# ===== #

## run/api: run the cmd/api application
.PHONY: run/api
run/api:
    go run ./cmd/api -db-dsn=${GREENLIGHT_DB_DSN}

## db/psql: connect to the database using psql
.PHONY: db/psql
db/psql:
    psql ${GREENLIGHT_DB_DSN}

## db/migrations/new name=$1: create a new database migration
.PHONY: db/migrations/new
db/migrations/new:
    @echo 'Creating migration files for ${name}...'
    migrate create -seq -ext=.sql -dir=./migrations ${name}

## db/migrations/up: apply all up database migrations
.PHONY: db/migrations/up
db/migrations/up: confirm
    @echo 'Running up migrations...'
    migrate -path ./migrations -database ${GREENLIGHT_DB_DSN} up

# ===== #
# QUALITY CONTROL
# ===== #

## audit: tidy dependencies and format, vet and test all code
.PHONY: audit
audit:
    @echo 'Tidying and verifying module dependencies...'
    go mod tidy
    go mod verify
    @echo 'Formatting code...'
    go fmt ./...
    @echo 'Vetting code...'
    go vet ./...
    staticcheck ./...
    @echo 'Running tests...'
    go test -race -vet=off ./...
```

Now that's done, all you need to do is type **make audit** to run these checks before you

commit any code changes into your version control system or build any binaries.

Let's give it a try. If you've been following along closely, the output should look very similar to this:

```
$ make audit
Tidying and verifying module dependencies...
go mod tidy
go: finding module for package gopkg.in/mail.v2
go: downloading gopkg.in/mail.v2 v2.3.1
go: found gopkg.in/mail.v2 in gopkg.in/mail.v2 v2.3.1
go mod verify
all modules verified
Formatting code...
go fmt ./...
Vetting code...
go vet ./...
staticcheck ./...
Running tests...
go test -race -vet=off ./...
?    greenlight.alexedwards.net/cmd/api      [no test files]
?    greenlight.alexedwards.net/cmd/examples/cors/preflight [no test files]
?    greenlight.alexedwards.net/cmd/examples/cors/simple   [no test files]
?    greenlight.alexedwards.net/internal/data              [no test files]
?    greenlight.alexedwards.net/internal/jsonlog          [no test files]
?    greenlight.alexedwards.net/internal/mailer           [no test files]
?    greenlight.alexedwards.net/internal/validator        [no test files]
```

That's looking good. The `go mod tidy` command resulted in some additional packages needing to be downloaded, but apart from that, all the checks completed successfully without any problems.

Additional Information

Testing

We covered the topic of testing in a lot of detail in the first *Let's Go* book, and the same principles apply again here. If you like, feel free to revisit the testing section of *Let's Go* and re-implement some of those same patterns in your API. For example, you might like to try:

- Creating an end-to-end test for the `GET /v1/healthcheck` endpoint to verify that the headers and response body are what you expect.
- Creating a unit-test for the `rateLimit()` middleware to confirm that it sends a `429 Too Many Requests` response after a certain number of requests.

- Creating an end-to-end integration test, using a test database instance, which confirms that the `authenticate()` and `requirePermission()` middleware work together correctly to allow or disallow access to specific endpoints.

Module Proxies and Vendoring

One of the risks of using third-party packages in your Go code is that the package repository may cease to be available. For example, the `httprouter` package plays a central part in our application, and if the author ever decided to delete it from GitHub it would cause us quite a headache to scramble and replace it with an alternative.

(I'm not suggesting this is likely to happen with `httprouter` — just using it as an example!)

Fortunately, Go provides two ways in which we can mitigate this risk: *module proxies* and *vendoring*.

Module proxies

In version 1.13, Go began to support *module proxies* (also known as *module mirrors*) by default. These are services which mirror source code from the original, authoritative, repositories (such as those hosted on GitHub, GitLab or BitBucket).

Go ahead and run the `go env` command on your machine to print out the settings for your Go operating environment. Your output should look similar to this:

```
$ go env
GO111MODULE=""
GOARCH="amd64"
GOBIN=""
GOCACHE="/home/alex/.cache/go-build"
GOENV="/home/alex/.config/go/env"
GOEXE=""
GOFLAGS=""
GOHOSTARCH="amd64"
GOHOSTOS="linux"
GOINSECURE=""
GOMODCACHE="/home/alex/go/pkg/mod"
GONOPROXY=""
GONOSUMDB=""
GOOS="linux"
GOPATH="/home/alex/go"
GOPRIVATE=""
GOPROXY="https://proxy.golang.org,direct"
GOROOT="/usr/local/go"
GOSUMDB="sum.golang.org"
GOTMPDIR=""
GOTOOLDIR="/usr/local/go/pkg/tool/linux_amd64"
GCCGO="gccgo"
AR="ar"
CC="gcc"
CXX="g++"
CGO_ENABLED="1"
GOMOD="/home/alex/Projects/greenlight/go.mod"
CGO_CFLAGS="-g -O2"
CGO_CPPFLAGS=""
CGO_CXXFLAGS="-g -O2"
CGO_FFLAGS="-g -O2"
CGO_LDFLAGS="-g -O2"
PKG_CONFIG="pkg-config"
GOGCCFLAGS=""
```

The important thing to look at here is the `GOPROXY` setting, which contains a comma-separated list of module mirrors. By default it has the following value:

```
GOPROXY="https://proxy.golang.org,direct"
```

The URL `https://proxy.golang.org` that we see here points to a *module mirror* maintained by the Go team at Google, containing copies of the source code from tens of thousands of open-source Go packages.

Whenever you fetch a package using the `go` command — either with `go get` or one of the `go mod *` commands — it will first attempt to retrieve the source code from this mirror.

If the mirror already has a stored copy of the source code for the required package and version number, then it will return this code immediately in a zip file. Otherwise, if it's not already stored, then the mirror will attempt to fetch the code from the authoritative repository, proxy it onwards to you, and store it for future use.

If the mirror can't fetch the code at all, then it will return an error response and the `go` tool will fall back to fetching a copy directly from the authoritative repository (thanks to the `direct` directive in the `GOPROXY` setting).

Using a module mirror as the first fetch location has a few benefits:

- The <https://proxy.golang.org> module mirror *typically* stores packages long-term, thereby providing a degree of protection in case the original repository disappears from the internet.
- It's not possible to override or delete a package once it's stored in the <https://proxy.golang.org> module mirror. This can help prevent any bugs or problems which might arise if a package author (or an attacker) releases an edited version of the package *with the same version number*.
- Fetching modules from the <https://proxy.golang.org> mirror can be **much faster** than getting them from the authoritative repositories.

In most cases, I would generally suggest leaving the `GOPROXY` setting with its default values.

But if you don't want to use the module mirror provided by Google, or you're behind a firewall that blocks it, there are other alternatives like <https://goproxy.io> and the Microsoft-provided <https://athens.azurefd.net> that you can try instead. Or you can even host your own module mirror using the open-source [Athens](#) and [goproxy](#) projects.

For example, if you wanted to switch to using <https://goproxy.io> as the primary mirror, then fall back to using <https://proxy.golang.org> as a secondary mirror, then fall back to a direct fetch, you could update your `GOPROXY` setting like so:

```
$ export GOPROXY=https://goproxy.io,https://proxy.golang.org,direct
```

Or if you want to disable module mirrors altogether, you can simply set the value to `direct` like so:

```
$ export GOPROXY=direct
```

Vendoring

Go's module mirror functionality is great, and I recommend using it. But it isn't a silver bullet for all developers and all projects.

For example, perhaps you don't want to use a module mirror provided by Google or another

third-party, but you also don't want the overhead of hosting your own mirror. Or maybe you need to routinely work in an environment without network access. In those scenarios you probably still want to mitigate the risk of a disappearing dependency, but using a module mirror isn't possible or appealing.

You should also be aware that the default `proxy.golang.org` module mirror doesn't absolutely guarantee that it will store a copy of the module forever. From [the FAQs](#):

proxy.golang.org does not save all modules forever. There are a number of reasons for this, but one reason is if proxy.golang.org is not able to detect a suitable license. In this case, only a temporarily cached copy of the module will be made available, and may become unavailable if it is removed from the original source and becomes outdated.

Additionally, if you need to come back to a 'cold' codebase in 5 or 10 years' time, will the `proxy.golang.org` module mirror still be available? Hopefully it will — but it's hard to say for sure.

So, for these reasons, it can still be sensible to *vendor* your project dependencies using the `go mod vendor` command. Vendoring dependencies in this way basically stores a complete copy of the source code for third-party packages in a `vendor` folder in your project.

Let's demonstrate how to do this. We'll start by adapting our `Makefile` to include a new `vendor` rule which calls the `go mod tidy`, `go mod verify` and `go mod vendor` commands, like so:

File: Makefile

```
...

# ===== #
# QUALITY CONTROL
# ===== #

## audit: tidy and vendor dependencies and format, vet and test all code
.PHONY: audit
audit: vendor
    @echo 'Formatting code...'
    go fmt ./...
    @echo 'Vetting code...'
    go vet ./...
    staticcheck ./...
    @echo 'Running tests...'
    go test -race -vet=off ./...

## vendor: tidy and vendor dependencies
.PHONY: vendor
vendor:
    @echo 'Tidying and verifying module dependencies...'
    go mod tidy
    go mod verify
    @echo 'Vendoring dependencies...'
    go mod vendor
```

As well as adding the `vendor` rule, there are a couple of other changes we've made here:

- We've removed the `go mod tidy` and `go mod verify` commands from the `audit` rule.
- We've added the `vendor` rule as a prerequisite to `audit`, which means it will automatically be run each time we execute the `audit` rule.

Just to be clear about what's going on behind-the-scenes here, let's quickly step through what will happen when we run `make vendor`:

- The `go mod tidy` command will make sure the `go.mod` and `go.sum` files list all the necessary dependencies for our project (and no unnecessary ones).
- The `go mod verify` command will verify that the dependencies stored in your module cache (located on your machine at `$GOPATH/pkg/mod`) match the cryptographic hashes in the `go.sum` file.
- The `go mod vendor` command will then copy the necessary source code from your module cache into a new `vendor` directory in your project root.

Let's try this out and run the new `vendor` rule like so:

```
$ make vendor
Tidying and verifying module dependencies...
go mod tidy
go mod verify
all modules verified
Vendoring dependencies...
go mod vendor
```

Once that's completed, you should see that a new **vendor** directory has been created containing copies of all the source code along with a **modules.txt** file. The directory structure in your **vendor** folder should look similar to this:

```
$ tree -L 3 ./vendor/
./vendor/
├── github.com
│   ├── felixge
│   │   └── httpsnoop
│   ├── go-mail
│   │   └── mail
│   ├── julienschmidt
│   │   └── httprouter
│   ├── lib
│   │   └── pq
│   └── golang.org
│       └── x
│           ├── crypto
│           └── time
├── gopkg.in
│   └── alexcesaro
│       └── quotedprintable.v3
└── modules.txt
```

Now, when you run a command such as **go run**, **go test** or **go build**, the **go** tool will recognize the presence of a **vendor** folder and *the dependency code in the vendor folder* will be used — rather than the code in the module cache on your local machine.

If you like, go ahead and try running the API application. You should find that everything compiles and continues to work just like before.

```
$ make run/api
go run ./cmd/api -db-dsn=postgres://greenlight:pa55word@localhost/greenlight
{"level":"INFO","time":"2021-04-18T13:18:29Z","message":"database connection pool established"}
{"level":"INFO","time":"2021-04-18T13:18:29Z","message":"starting server","properties":{"addr":":4000","env":"development"}}}
```

Note: If you want to confirm that it's really the vendored dependencies being used, you can run **go clean -modcache** to remove *everything* from your local module cache. When you run the API again, you should find that it still starts up correctly *without* needing to re-fetch the dependencies from the Go module mirror.

Because all the dependency source code is now stored in your project repository itself, it's easy to check it into Git (or an alternative version control system) alongside the rest of your code. This is reassuring because it gives you *complete ownership* of all the code used to build and run your applications, kept under version control.

The downside of this, of course, is that it adds size and bloat to your project repository. This is of particular concern in projects that have a lot of dependencies and the repository will be cloned *a lot*, such as projects where a CI/CD system clones the repository with each new commit.

Let's also take a quick look in the `vendor/modules.txt` file that was created. If you've been following along it should look similar to this:

File: vendor/modules.txt

```
# github.com/felixge/httpsnoop v1.0.2
## explicit; go 1.13
github.com/felixge/httpsnoop
# github.com/go-mail/mail/v2 v2.3.0
## explicit
github.com/go-mail/mail/v2
# github.com/julienschmidt/httprouter v1.3.0
## explicit; go 1.7
github.com/julienschmidt/httprouter
# github.com/lib/pq v1.10.2
## explicit; go 1.13
github.com/lib/pq
github.com/lib/pq/oid
github.com/lib/pq/scram
# golang.org/x/crypto v0.0.0-20210817164053-32db794688a5
## explicit; go 1.17
golang.org/x/crypto/bcrypt
golang.org/x/crypto/blowfish
# golang.org/x/time v0.0.0-20210723032227-1f47c861a9ac
## explicit
golang.org/x/time/rate
# gopkg.in/alexcesaro/quotedprintable.v3 v3.0.0-20150716171945-2caba252f4dc
## explicit
gopkg.in/alexcesaro/quotedprintable.v3
# gopkg.in/mail.v2 v2.3.1
## explicit
```

This `vendor/modules.txt` file is essentially a *manifest* of the vendored packages and their version numbers. When vendoring is being used, the `go` tool will check that the module version numbers in `modules.txt` are consistent with the version numbers in the `go.mod` file. If there's any inconsistency, then the `go` tool will report an error.

Note: It's important to point out that there's no easy way to verify that the *checksums of the vendored dependencies* match the checksums in the `go.sum` file. Or, in other words, there's no equivalent to `go mod verify` which works *directly* on the contents of the `vendor` folder.

To mitigate that, it's a good idea to run *both* `go mod verify` and `go mod vendor` regularly. Using `go mod verify` will verify that the dependencies in your module cache match the `go.sum` file, and `go mod vendor` will copy those same dependencies from the module cache into your `vendor` folder. This is one of the reasons why our `make vendor` rule is setup to run both commands, and why we've also included it as a prerequisite to the `make audit` rule.

Lastly, you should avoid making any changes to the code in the `vendor` directory. Doing so can potentially cause confusion (because the code would no longer be consistent with the original version of the source code) and — besides — running `go mod vendor` will overwrite any changes you make each time you run it. If you need to change the code for a dependency, it's far better to fork it and import the forked version instead.

Vendoring new dependencies

In the next section of the book we're going to deploy our API application to the internet with `Caddy` as a reverse-proxy in-front of it. This means that, as far as our API is concerned, all the requests it receives will be coming from a single IP address (the one running the `Caddy` instance). In turn, that will cause problems for our rate limiter middleware which limits access based on IP address.

Fortunately, like most other reverse proxies, `Caddy` adds an `X-Forwarded-For` header to each request. This header will contain the *real IP address* for the client.

Although we could write the logic to check for the presence of an `X-Forwarded-For` header and handle it ourselves, I recommend using the `realip` package to help with this. This package retrieves the client IP address from any `X-Forwarded-For` or `X-Real-IP` headers, falling back to use `r.RemoteAddr` if neither of them are present.

If you're following along, go ahead and install the latest version of `realip` using the `go get` command:


```
$ go get github.com/tomasen/realip@latest
go: downloading github.com/tomasen/realip v0.0.0-20180522021738-f0c99a92ddce
go get: added github.com/tomasen/realip v0.0.0-20180522021738-f0c99a92ddce
```

Then open up the `cmd/api/middleware.go` file and update the `rateLimit()` middleware to use this package like so:

File: cmd/api/middleware.go

```
package main

import (
    "errors"
    "expvar"
    "fmt"
    "net/http"
    "strconv"
    "strings"
    "sync"
    "time"

    "greenlight.alexedwards.net/internal/data"
    "greenlight.alexedwards.net/internal/validator"

    "github.com/felixge/httpsnoop"
    "github.com/tomasen/realip" // New import
    "golang.org/x/time/rate"
)

...

func (app *application) ratelimit(next http.Handler) http.Handler {
    ...

    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        if app.config.limiter.enabled {
            // Use the realip.FromRequest() function to get the client's real IP address.
            ip := realip.FromRequest(r)

            mu.Lock()

            if _, found := clients[ip]; !found {
                clients[ip] = &client{
                    limiter: rate.NewLimiter(rate.Limit(app.config.limiter.rps), app.config.limiter.burst),
                }
            }

            clients[ip].lastSeen = time.Now()

            if !clients[ip].limiter.Allow() {
                mu.Unlock()
                app.rateLimitExceededResponse(w, r)
                return
            }

            mu.Unlock()
        }

        next.ServeHTTP(w, r)
    })
}

...
```

If you try to run the API application again now, you should receive an error message similar to this:

```
$ make run/api
go: inconsistent vendoring in /home/alex/Projects/greenlight:
    github.com/tomasen/realip@v0.0.0-20180522021738-f0c99a92ddce: is explicitly
        required in go.mod, but not marked as explicit in vendor/modules.txt

    To ignore the vendor directory, use -mod=readonly or -mod=mod.
    To sync the vendor directory, run:
        go mod vendor
make: *** [Makefile:24: run/api] Error 1
```

Essentially what's happening here is that Go is looking for the `github.com/tomasen/realip` package in our `vendor` directory, but at the moment that package doesn't exist in there.

To solve this, you'll need to manually run either the `make vendor` or `make audit` commands, like so:

```
$ make vendor
Tidying and verifying module dependencies...
go mod tidy
go: finding module for package github.com/tomasen/realip
go: downloading github.com/tomasen/realip v0.0.0-20180522021738-f0c99a92ddce
go: found github.com/tomasen/realip in github.com/tomasen/realip v0.0.0-20180522021738-f0c99a92ddce
go mod verify
all modules verified
Vendoring dependencies...
go mod vendor
```

Once that's done, everything should work correctly again:

```
$ make run/api
go run ./cmd/api -db-dsn=postgres://greenlight:pa55word@localhost/greenlight
{"level":"INFO","time":"2021-04-18T13:33:33Z","message":"database connection pool established"}
{"level":"INFO","time":"2021-04-18T13:33:33Z","message":"starting server","properties":{"addr":":4000","env":"development"}}
```

Additional Information

The `./...` pattern

Most of the `go` tools support the `./...` wildcard pattern, like `go fmt ./...`, `go vet ./...` and `go test ./...`. This pattern matches the current directory and all sub-directories, *excluding the `vendor` directory*.

Generally speaking, this is useful because it means that we're not formatting, vetting or testing the code in our `vendor` directory unnecessarily — and our `make audit` rule won't fail

due to any problems that might exist *within* those vendored packages.

Building Binaries

So far we've been running our API using the `go run` command (or more recently, `make run/api`). But in this chapter we're going to focus on explaining how to *build an executable binary* that you can distribute and run on other machines without needing the Go toolchain installed.

To build a binary we need to use the `go build` command. As a simple example, usage looks like this:

```
$ go build -o=./bin/api ./cmd/api
```

When we run this command, `go build` will *compile* the `cmd/api` package (and any dependent packages) into files containing machine code, and then *link* these together to form executable binary. In the command above, the executable binary will be output to `./bin/api`.

For convenience, let's add a new `build/api` rule to our makefile which runs this command, like so:

```
File: Makefile

...

# ===== #
# BUILD
# ===== #

## build/api: build the cmd/api application
.PHONY: build/api
build/api:
@echo 'Building cmd/api...'
go build -o=./bin/api ./cmd/api
```

Once that's done, go ahead and execute the `make build/api` rule. You should see that an executable binary file gets created at `./bin/api`.

```
$ make build/api
Building cmd/api...
go build -o=./bin/api ./cmd/api
$ ls -l ./bin/
total 10228
-rwxrwxr-x 1 alex alex 10470419 Apr 18 16:05 api
```

And you should be able to run this executable to start your API application, passing in any command-line flag values as necessary. For example:

```
$ ./bin/api -port=4040 -db-dsn=postgres://greenlight:pa55word@localhost/greenlight
{"level":"INFO","time":"2021-04-18T14:06:11Z","message":"database connection pool established"}
{"level":"INFO","time":"2021-04-18T14:06:11Z","message":"starting server","properties":{"addr":":4040","env":"development"}}
```

Reducing binary size

If you take a closer look at the executable binary you'll see that it weighs in at 10470419 bytes (about 10.5MB).

```
$ ls -l ./bin/api
-rwxrwxr-x 1 alex alex 10470419 Apr 18 16:05 ./bin/api
```

Note: If you're following along, your binary size may be slightly different. It depends on your operating system, and the exact version of Go and the third-party dependencies that you're using.

It's possible to reduce the binary size by around 25% by instructing the Go linker to strip the [DWARF](#) debugging information and [symbol table](#) from the binary. We can do this as part of the `go build` command by using the *linker flag* `-ldflags="-s"` as follows:

File: Makefile

```
...

# ===== #
# BUILD
# ===== #

## build/api: build the cmd/api application
.PHONY: build/api
build/api:
@echo 'Building cmd/api...'
go build -ldflags='-s' -o=./bin/api ./cmd/api
```

If you run `make build/api` again, you should now find that the size of the binary is reduced to under 8 MB.

```
$ make build/api
Building cmd/api...
go build -ldflags='-s' -o=./bin/api ./cmd/api
$ ls -l ./bin/api
-rwxrwxr-x 1 alex alex 7618560 Apr 18 16:08 ./bin/api
```

It's important to be aware that stripping the DWARF information and symbol table will make it harder to debug an executable using a tool like [Delve](#) or [gdb](#). But, generally, it's not often that you'll need to do this — and there's even an [open proposal](#) from Rob Pike to make omitting DWARF information the default behavior of the linker in the future.

Cross-compilation

By default, the `go build` command will output a binary suitable for use on your *local machine's operating system and architecture*. But it also supports cross-compilation, so you can generate a binary suitable for use on a different machine. This is particularly useful if you're developing on one operating system and deploying on another.

To see a list of all the operating system/architecture combinations that Go supports, you can run the `go tool dist list` command like so:

```
$ go tool dist list
aix/ppc64
android/386
android/amd64
android/arm
android/arm64
darwin/amd64
...
```

And you can specify the operating system and architecture that you want to create the binary for by setting `GOOS` and `GOARCH` environment variables when running `go build`. For example:

```
$ GOOS=linux GOARCH=amd64 go build {args}
```

In the next section of the book, we're going to walk through how to deploy an executable binary on an Ubuntu Linux server hosted by Digital Ocean. For this we'll need a binary which is designed to run on a machine with a `linux/amd64` OS and architecture combination.

So let's update our `make build/api` rule so that it creates two binaries — one for use on your local machine, and another for deploying to the Ubuntu Linux server.

File: Makefile

```
...

# ===== #
# BUILD
# ===== #

## build/api: build the cmd/api application
.PHONY: build/api
build/api:
@echo 'Building cmd/api...'
go build -ldflags='-s' -o=./bin/api ./cmd/api
GOOS=linux GOARCH=amd64 go build -ldflags='-s' -o=./bin/linux_amd64/api ./cmd/api
```

If you're following along, go ahead and run `make build/api` again.

You should see that two binaries are now created — with the cross-compiled binary located under the `./bin/linux_amd64` directory, like so:

```
$ make build/api
Building cmd/api...
go build -ldflags='-s' -o=./bin/api ./cmd/api
GOOS=linux GOARCH=amd64 go build -ldflags='-s' -o=./bin/linux_amd64/api ./cmd/api
$ tree ./bin
./bin
├── api
└── linux_amd64
    └── api
```

As a general rule, you probably don't want to commit your Go binaries into version control alongside your source code as they will significantly inflate the size of your repository.

So, if you're following along, let's quickly add an additional rule to the `.gitignore` file which instructs Git to ignore the contents of the `bin` directory.

```
$ echo 'bin/' >> .gitignore
$ cat .gitignore
.envrc
bin/
```

Additional Information

Build caching

It's important to note that the `go build` command caches build output in the Go *build*

cache. This cached output will be reused again in future builds where appropriate, which can significantly speed up the overall build time for your application.

If you're not sure where your build cache is, you can check by running the `go env GOCACHE` command:

```
$ go env GOCACHE
/home/alex/.cache/go-build
```

You should also be aware that the build cache does not automatically detect any changes to C libraries that your code imports with `cgo`. So, if you've changed a C library since the last build, you'll need to use the `-a` flag to force all packages to be rebuilt when running `go build`. Alternatively, you could use `go clean` to purge the cache:

```
$ go build -a -o=/bin/foo ./cmd/foo      # Force all packages to be rebuilt
$ go clean -cache                        # Remove everything from the build cache
```

Note: If you ever run `go build` on a non-`main` package, the build output will be stored in the build cache so it can be reused, but no executable will be produced.

Managing and Automating Version Numbers

Right at the start of this book, we hard-coded the version number for our application as the constant `"1.0.0"` in the `cmd/api/main.go` file.

In this chapter, we're going to take steps to make it easier to view and manage this version number, and also explain how you can generate version numbers automatically based on Git commits and integrate them into your application.

Displaying the version number

Let's start by updating our application so that we can easily check the version number by running the binary with a `-version` command-line flag, similar to this:

```
$ ./bin/api -version
Version:      1.0.0
```

Conceptually, this is fairly straightforward to implement. We need to define a boolean `version` command-line flag, check for this flag on startup, and then print out the version number and exit the application if necessary.

If you're following along, go ahead and update your `cmd/api/main.go` file like so:

```
File: cmd/api/main.go

package main

import (
    "context"
    "database/sql"
    "expvar"
    "flag"
    "fmt" // New import
    "os"
    "runtime"
    "strings"
    "sync"
    "time"

    "greenlight.alexedwards.net/internal/data"
    "greenlight.alexedwards.net/internal/jsonlog"
    "greenlight.alexedwards.net/internal/mailer"
```

```

    _ "github.com/lib/pq"
)

const version = "1.0.0"

...

func main() {
    var cfg config

    flag.IntVar(&cfg.port, "port", 4000, "API server port")
    flag.StringVar(&cfg.env, "env", "development", "Environment (development|staging|production)")

    flag.StringVar(&cfg.db.dsn, "db-dsn", "", "PostgreSQL DSN")

    flag.IntVar(&cfg.db.maxOpenConns, "db-max-open-conns", 25, "PostgreSQL max open connections")
    flag.IntVar(&cfg.db.maxIdleConns, "db-max-idle-conns", 25, "PostgreSQL max idle connections")
    flag.StringVar(&cfg.db.maxIdleTime, "db-max-idle-time", "15m", "PostgreSQL max connection idle time")

    flag.BoolVar(&cfg.limiter.enabled, "limiter-enabled", true, "Enable rate limiter")
    flag.Float64Var(&cfg.limiter.rps, "limiter-rps", 2, "Rate limiter maximum requests per second")
    flag.IntVar(&cfg.limiter.burst, "limiter-burst", 4, "Rate limiter maximum burst")

    flag.StringVar(&cfg.smtp.host, "smtp-host", "smtp.mailtrap.io", "SMTP host")
    flag.IntVar(&cfg.smtp.port, "smtp-port", 25, "SMTP port")
    flag.StringVar(&cfg.smtp.username, "smtp-username", "0abf276416b183", "SMTP username")
    flag.StringVar(&cfg.smtp.password, "smtp-password", "d8672aa2264bb5", "SMTP password")
    flag.StringVar(&cfg.smtp.sender, "smtp-sender", "Greenlight <no-reply@greenlight.alexedwards.net>", "SMTP sender")

    flag.Func("cors-trusted-origins", "Trusted CORS origins (space separated)", func(val string) error {
        cfg.cors.trustedOrigins = strings.Fields(val)
        return nil
    })

    // Create a new version boolean flag with the default value of false.
    displayVersion := flag.Bool("version", false, "Display version and exit")

    flag.Parse()

    // If the version flag value is true, then print out the version number and
    // immediately exit.
    if *displayVersion {
        fmt.Printf("Version:\t%s\n", version)
        os.Exit(0)
    }

    ...
}

...

```

OK, let's try this out. Go ahead and re-build the executable binaries using `make build/api`, then run the `./bin/api` binary with the `-version` flag.

You should find that it prints out the version number and then exits, similar to this:

```
$ make build/api
Building cmd/api...
go build -ldflags="-s" -o="./bin/api" ./cmd/api
GOOS=linux GOARCH=amd64 go build -ldflags="-s" -o="./bin/linux_amd64/api" ./cmd/api

$ ./bin/api -version
Version:      1.0.0
```

Remember: Boolean command-line flags *without a value* are interpreted as having the value `true`. So running our application with `-version` is the same as running it with `-version=true`.

Including the build time

Let's extend this further, so that the `-version` flag forces our application to print out the version number *and* the exact time that the executable binary was built. Similar to this:

```
$ ./bin/api -version
Version:      1.0.0
Build time:   2021-02-15T13:54:14+01:00
```

This is a really interesting change, because the build time is a completely dynamic value and *outside the control of our application code*. We don't know the build time *in advance* of calling `go build`, and therefore it's not something that we can define in our source code in the normal way.

To make this work, we'll need to use a special feature of `go build` — the `-X` linker flag which allows you to assign or 'burn-in' a value to a variable when running `go build`. Specifically, we'll generate a string containing the current date and time in our makefile, and then 'burn this in' to the binary as part of our `make build/api` rule.

I'll demonstrate.

Open up the `cmd/api/main.go` file and update it to include and print out a new `buildTime` variable, like so:

File: cmd/api/main.go

```
package main

...

const version = "1.0.0"

// Create a buildTime variable to hold the executable binary build time. Note that this
// must be a string type, as the -X linker flag will only work with string variables.
var buildTime string

...

func main() {
    var cfg config

    flag.IntVar(&cfg.port, "port", 4000, "API server port")
    flag.StringVar(&cfg.env, "env", "development", "Environment (development|staging|production)")

    flag.StringVar(&cfg.db.dsn, "db-dsn", "", "PostgreSQL DSN")

    flag.IntVar(&cfg.db.maxOpenConns, "db-max-open-conns", 25, "PostgreSQL max open connections")
    flag.IntVar(&cfg.db.maxIdleConns, "db-max-idle-conns", 25, "PostgreSQL max idle connections")
    flag.StringVar(&cfg.db.maxIdleTime, "db-max-idle-time", "15m", "PostgreSQL max connection idle time")

    flag.BoolVar(&cfg.limiter.enabled, "limiter-enabled", true, "Enable rate limiter")
    flag.Float64Var(&cfg.limiter.rps, "limiter-rps", 2, "Rate limiter maximum requests per second")
    flag.IntVar(&cfg.limiter.burst, "limiter-burst", 4, "Rate limiter maximum burst")

    flag.StringVar(&cfg.smtp.host, "smtp-host", "smtp.mailtrap.io", "SMTP host")
    flag.IntVar(&cfg.smtp.port, "smtp-port", 25, "SMTP port")
    flag.StringVar(&cfg.smtp.username, "smtp-username", "0abf276416b183", "SMTP username")
    flag.StringVar(&cfg.smtp.password, "smtp-password", "d8672aa2264bb5", "SMTP password")
    flag.StringVar(&cfg.smtp.sender, "smtp-sender", "Greenlight <no-reply@greenlight.alexedwards.net>", "SMTP sender")

    flag.Func("cors-trusted-origins", "Trusted CORS origins (space separated)", func(val string) error {
        cfg.cors.trustedOrigins = strings.Fields(val)
        return nil
    })

    displayVersion := flag.Bool("version", false, "Display version and exit")

    flag.Parse()

    if *displayVersion {
        fmt.Printf("Version:\t%s\n", version)
        // Print out the contents of the buildTime variable.
        fmt.Printf("Build time:\t%s\n", buildTime)
        os.Exit(0)
    }

    ...
}
```

And then we need to make a couple of changes to our **Makefile**. Specifically, we need to:

1. Use the unix **date** command to generate the current time and store it in a **current_time** variable.

2. Update the `build/api` rule to ‘burn in’ the current time to our `main.buildTime` variable using the `-X` linker flag.

Like this:

File: Makefile

```
...

# ===== #
# BUILD
# ===== #

current_time = $(shell date --iso-8601=seconds)

## build/api: build the cmd/api application
.PHONY: build/api
build/api:
@echo 'Building cmd/api...'
go build -ldflags='-s -X main.buildTime=${current_time}' -o=./bin/api ./cmd/api
GOOS=linux GOARCH=amd64 go build -ldflags='-s -X main.buildTime=${current_time}' -o=./bin/linux_amd64/api ./cmd/api
```

Important: Old versions of the `date` command may not support the `--iso-8601` flag. If this affects you, then you can swap the code above to use the command `date -u +"%Y-%m-%dT%H:%M:%SZ"` instead.

If you re-build the binaries and run them with the `-version` flag, they should now print the exact time that they were built alongside the version number. Similar to this:

```
$ make build/api
Building cmd/api...
go build -ldflags='-s -X main.buildTime=2021-04-18T18:21:40+02:00' -o=./bin/api ./cmd/api
GOOS=linux GOARCH=amd64 go build -ldflags='-s -X main.buildTime=2021-04-18T18:21:40+02:00' -o=./bin/linux_amd64/api ./cmd/api

$ ./bin/api -version
Version:      1.0.0
Build time:   2021-04-18T18:21:40+02:00
```

Note that if you execute the application *without* burning in a `buildTime` value, then `buildTime` will retain its default value as set in the application code (which in our case is the empty string `""`). For example, when using `go run`:

```
$ go run ./cmd/api -version
Version:      1.0.0
Build time:
```

The lines in our Makefile are getting a bit long, so let’s quickly break out the linker flags into

a reusable variable, like so:

File: Makefile

```
...

# ===== #
# BUILD
# ===== #

current_time = $(shell date --iso-8601=seconds)
linker_flags = '-s -X main.buildTime=${current_time}'

## build/api: build the cmd/api application
.PHONY: build/api
build/api:
@echo 'Building cmd/api...'
go build -ldflags=${linker_flags} -o=./bin/api ./cmd/api
GOOS=linux GOARCH=amd64 go build -ldflags=${linker_flags} -o=./bin/linux_amd64/api ./cmd/api
```

Automated version numbering with Git

If you're using Git to version control your source code, then it's possible to extend the ideas we've talked about in this chapter to create *automated version numbers* for your application based on your Git commits.

Important: This part of the book is only really relevant if you use Git. If you don't, then that's absolutely fine and you can skip ahead to the next chapter with no issues.

Otherwise, if you're following along (and haven't done it already), please go ahead and initialize a new Git repository in the root of your project directory:

```
$ git init
Initialized empty Git repository in /home/alex/Projects/greenlight/.git/
```

Then make a new commit containing all the files in your project directory, like so:

```
$ git add .
$ git commit -m "Initial commit"
```

If you look at your commit history using the `git log` command, you'll see the *hash* for this commit.

```
$ git log
commit 0f8573aab79f3cbee17d39901d5cb200121c7ed1 (HEAD -> master)
Author: Alex Edwards <alex@alexedwards.net>
Date: Sun Apr 18 18:29:48 2021 +0200

Initial commit
```

In my case the commit hash is `0f8573aab79f3cbee17d39901d5cb200121c7ed1` — but yours will be a different value.

It's also possible to use the `git describe` command to get a 'human-readable' descriptor for the current state of the repository. Like so:

```
$ git describe --always --dirty
0f8573a
```

In this case, we can see that `git describe` returns an *abbreviated version of the commit hash*. We're also using the `--dirty` flag, which means that the descriptor will be suffixed with `"-dirty"` if there are any uncommitted changes in the repository. For example `0f8573a-dirty`.

Let's update our project so that the output from `git describe` is burnt-in to our binaries as the application 'version number' when building the executable.

To do this, the first thing that we need to do is switch the `version` constant in our `cmd/api/main.go` file to be a variable, because it's not possible to 'burn-in' a value to a constant.

```
File: cmd/api/main.go

package main

...

// Remove the hardcoded version number and make version a variable instead of a constant.
var (
    buildTime string
    version    string
)

...
```

And then let's update our `Makefile` to burn-in the `git describe` output to the `version` variable in our application, using the same pattern that we did for the build time. Like so:

File: Makefile

```
...

# ===== #
# BUILD
# ===== #

current_time = $(shell date --iso-8601=seconds)
git_description = $(shell git describe --always --dirty)
linker_flags = '-s -X main.buildTime=${current_time} -X main.version=${git_description}'

## build/api: build the cmd/api application
.PHONY: build/api
build/api:
@echo 'Building cmd/api...'
go build -ldflags=${linker_flags} -o=./bin/api ./cmd/api
GOOS=linux GOARCH=amd64 go build -ldflags=${linker_flags} -o=./bin/linux_amd64/api ./cmd/api
```

At this point, let's quickly run `git describe` again:

```
$ git describe --always --dirty
0f8573a-dirty
```

This output makes sense — the hash of the last commit was `0f8573a...`, and the `-dirty` suffix indicates that our repository now contains uncommitted changes (we've altered the `cmd/api/main.go` and `Makefile` files since the previous commit).

Let's commit these recent changes, run `make build/api` again, and then check the version number in our binaries. Like so:

```
$ git add .
$ git commit -m "Generate version number automatically"
[master 76c976a] Generate version number automatically
 2 files changed, 6 insertions(+), 7 deletions(-)

$ make build/api
Building cmd/api...
go build -ldflags='-s -X main.buildTime=2021-04-18T18:43:32+02:00 -X main.version=76c976a' -o=./bin/api ./cmd/api
GOOS=linux GOARCH=amd64 go build -ldflags='-s -X main.buildTime=2021-04-18T18:43:32+02:00 -X main.version=76c976a' -o=./bin/linux_amd64/api ./cmd/api

$ ./bin/api -version
Version:      76c976a
Build time:   2021-04-18T18:43:32+02:00
```

We can see that our binary is now reporting that it was been built from a *clean version of the repository with the commit hash 76c976a*. Let's cross check this against the `git log` output for the project:

```
$ git log
commit 76c976a07d0fce3d813f3a4748574f81b1e24226 (HEAD -> master)
Author: Alex Edwards <alex@alexedwards.net>
Date: Sun Apr 18 18:39:17 2021 +0200

    Generate version number automatically

commit 0f8573aab79f3cbee17d39901d5cb200121c7ed1
Author: Alex Edwards <alex@alexedwards.net>
Date: Sun Apr 18 18:29:48 2021 +0200

    Initial commit
```

That's looking good — the commit hash in our Git history aligns perfectly with our application version number. And that means it's now easy for us to identify exactly what code a particular binary contains — all we need to do is run the binary with the **-version** flag and then cross-reference it against the Git repository history.

Using Git tags

In some projects you may want to annotate certain Git commits with **tags**, often to denote a formal release number.

To illustrate this, let's add the **v1.0.0** tag to our latest commit like so:

```
$ git tag v1.0.0
$ git log
commit 76c976a07d0fce3d813f3a4748574f81b1e24226 (HEAD -> master, tag: v1.0.0)
Author: Alex Edwards <alex@alexedwards.net>
Date: Sun Apr 18 18:39:17 2021 +0200

    Generate version number automatically

...
```

If you're tagging your commits in this way, then it's a good idea to use the **--tags** and **--long** flags when calling **git describe** to generate the version number for your application. This will force the output to be in the following format:

```
{tag}-{number of additional commits}-g{abbreviated commit hash}
```

Let's take a quick look at what this currently looks like for our project:

```
$ git describe --always --dirty --tags --long
v1.0.0-0-g76c976a
```

So, in this case, we can see from the version number that the repository is based on **v1.0.0**, has 0 additional commits on top of that, and the latest commit hash is **76c976a**. The **g** character which prefixes the commit hash stands for 'git', and exists to help distinguish the hash from any hashes generated by other version-control systems.

Let's update our **Makefile** to use these additional flags when generating the version number, like so:

```
File: Makefile

...

# ===== #
# BUILD
# ===== #

current_time = $(shell date --iso-8601=seconds)
git_description = $(shell git describe --always --dirty --tags --long)
linker_flags = '-s -X main.buildTime=${current_time} -X main.version=${git_description}'

...
```

Then go ahead and commit this change, rebuild the binaries, and check the version number. Your output should look similar to this:

```
$ git add .
$ git commit -m "Use --tags and --long flags when generating version number"
$ make build/api
$ ./bin/api -version
Version:      v1.0.0-1-gf27fd0f
Build time:   2021-04-18T18:54:05+02:00
```

The version number has now been changed to **v1.0.0-1-gf27fd0f**, indicating that the binary was built using the repository code from commit **f27fd0f**, which is 1 commit ahead of the **v1.0.0** tag.

Deployment and Hosting

In this final section of the book we're going to look at how to deploy our API application to a production server and expose it on the internet.

Every project and project team will have different technical and business needs in terms of hosting and deployment, so it's impossible to lay out a one-size-fits-all approach here.

To make the content in this section as widely-applicable and portable as possible, we'll focus on hosting the application on a self-managed Linux server (something provided by a myriad of hosting companies worldwide) and using standard Linux tooling to manage server configuration and deployment.

We'll also be automating the server configuration and deployment process as much as possible, so that it's easy to make *continuous deployments* and possible to *replicate the server* again in the future if you need to.

If you're planning to follow along, we'll be using [Digital Ocean](#) as the hosting provider in this book. Digital Ocean isn't free, but it's good value and the cost of running a server starts at \$5 USD per month. If you don't want to use Digital Ocean, you should be able to follow basically the same approach that we outline here with any other Linux hosting provider.

In terms of infrastructure and architecture, we'll run everything on a single Ubuntu Linux server. Our stack will consist of a PostgreSQL database and the executable binary for our Greenlight API, operating in much the same way that we have seen so far in this book. But in addition to this, we'll also run [Caddy](#) as a *reverse proxy* in front of the Greenlight API.

Using Caddy has a couple of benefits. It will automatically handle and terminate HTTPS connections for us — including automatically generating and managing TLS certificates via [Let's Encrypt](#) — and we can also use Caddy to easily restrict internet access to our metrics endpoint.

In this section you'll learn how to:

- Commission an Ubuntu Linux server running on Digital Ocean to host your application.
- Automate the configuration of the server — including creating user accounts, configuring the firewall and installing necessary software.
- Automate the process of updating your application and deploying changes to the server.

- How to run your application as a background service using [systemd](#), as a non-root user.
- Use Caddy as a reverse proxy in front of your application to automatically manage TLS certificates and handle HTTPS connections.

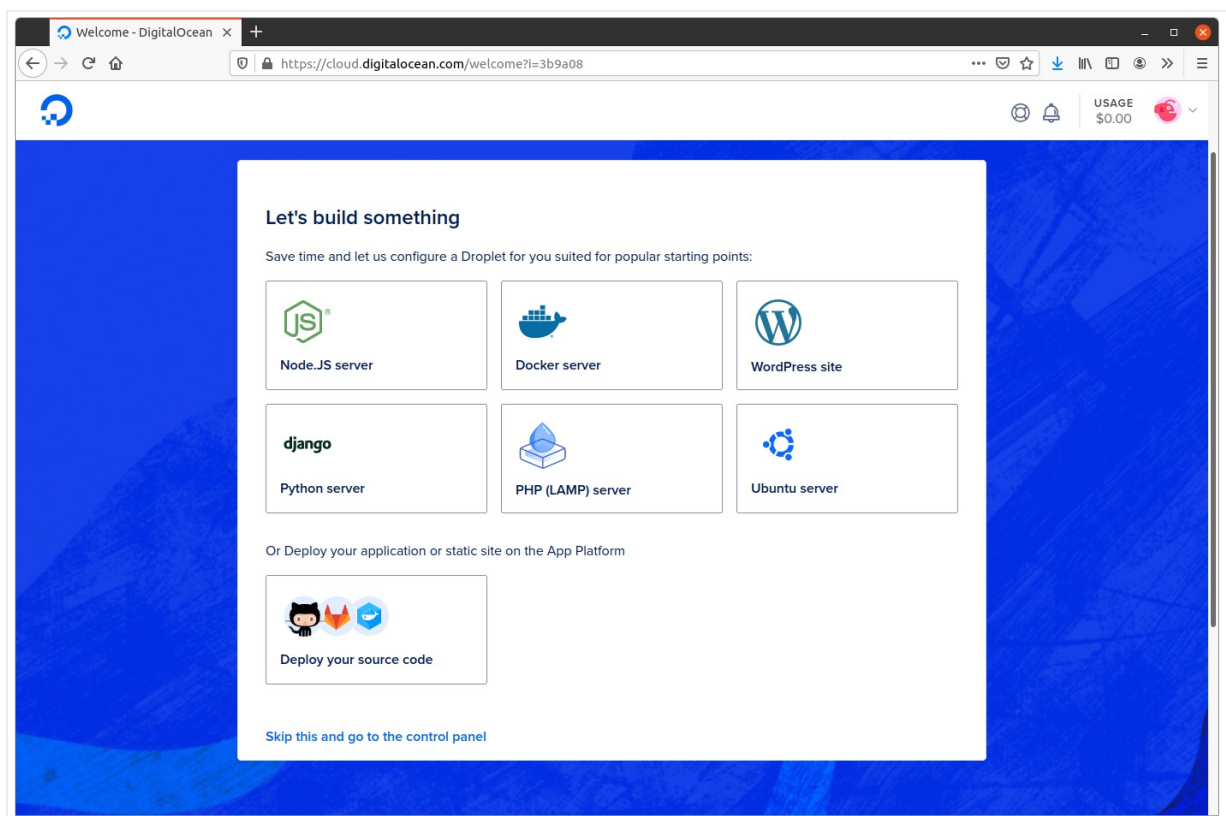
Creating a Digital Ocean Droplet

The first thing that we need to do is commission a server on Digital Ocean to host our application.

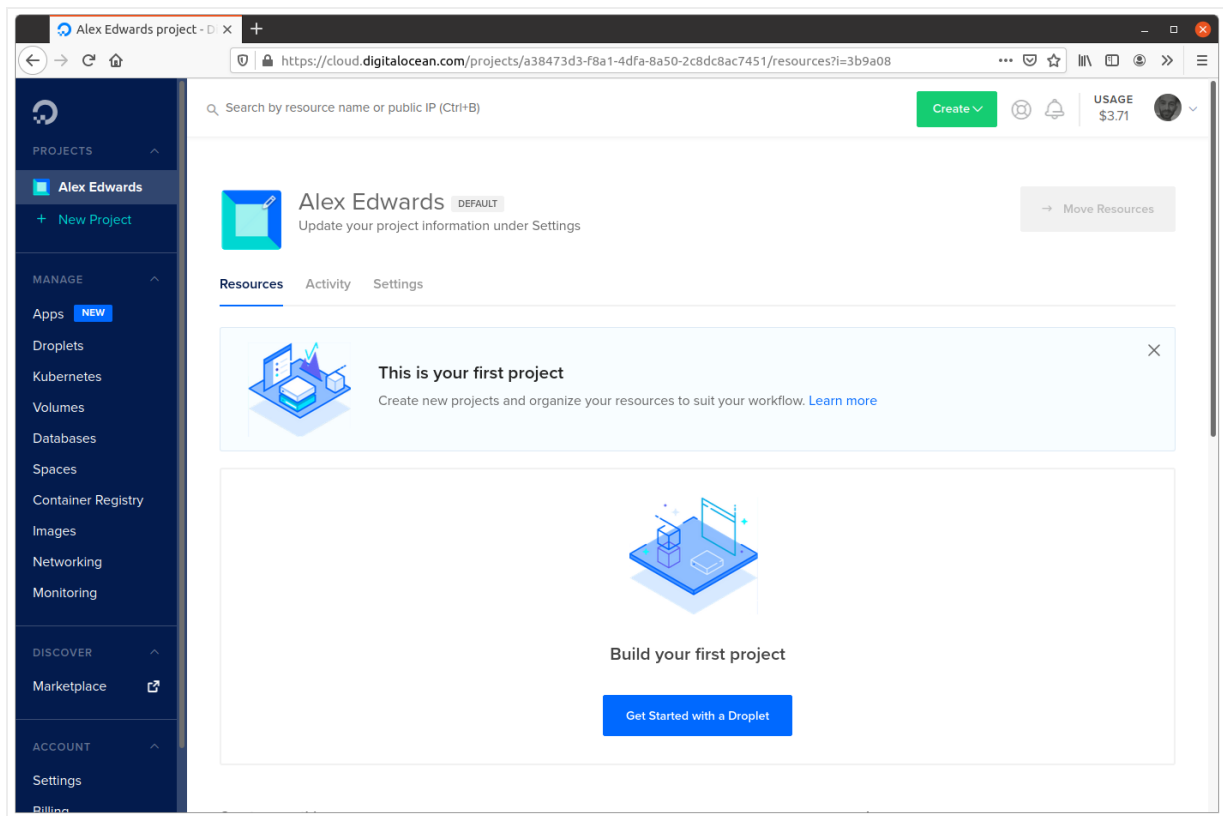
Strictly speaking, what we're going to commission is actually a *virtual machine* known in Digital Ocean terminology as a *droplet*.

If you'd like to follow along with this stage of the book, you'll need to [register for a Digital Ocean account](#) if you don't already have one. As part of the registration process, you'll be prompted confirm your email address and then add a minimum of \$5 USD pre-paid credit to your account using either a credit/debit card or PayPal.

Once you've completed the registration and added the credit, you should find yourself presented with an introductory screen similar to this:



Click the “Skip this” link at the bottom of the screen to go straight to your account control panel, which should look like this:



Creating a SSH key

In order to log in to droplets in your Digital Ocean account you'll need a *SSH keypair*.

Suggestion: If you're unfamiliar with SSH, SSH keys, or [public-key cryptography](#) generally, then I recommend reading through the first half of [this guide](#) to get an overview before continuing.

If you already have a SSH keypair that you're happy to use for this purpose, then that's great, and you can skip ahead to the next section.

But if you don't, you'll need to create a keypair using the [ssh-keygen](#) command on your local machine. Similar to this:

```
$ ssh-keygen -t rsa -b 4096 -C "greenlight@greenlight.alexedwards.net" -f $HOME/.ssh/id_rsa_greenlight
Generating public/private rsa key pair.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/alex/.ssh/id_rsa_greenlight
Your public key has been saved in /home/alex/.ssh/id_rsa_greenlight.pub
The key fingerprint is:
SHA256:/ii7Wo90BdsoaMU1UAajmd/5vBSeVmbRtf7Qtoww8YM greenlight@greenlight.alexedwards.net
The key's randomart image is:
+---[RSA 4096]-----+
|      ++=  ....  |
|      =+  . . .  |
|      +o  . o o .  |
|      + . B . B o  |
|      o o S + E * + |
|      . o = + =o|
|      o o B  ..|
|      o.+ = .  |
|      ..++o o  |
+----[SHA256]-----+
```

This will generate two new files in your `$HOME/.ssh` folder:

- The `$HOME/.ssh/id_rsa_greenlight` file contains your *private key*. Make sure to keep this secure, because anyone who has access to it will be able to impersonate you.
- The `$HOME/.ssh/id_rsa_greenlight.pub` file contains your *public key*. We'll upload a copy of this public key to Digital Ocean.

If you open your public key file in a text editor you should see that the content looks similar to this (line breaks added for readability):

```
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQDdAjtGVtW4R11nDvFI7PgE712+xiXm7v6slsLMmJ0IzpkN+f
HfDn4Nx/QSGnFbRift2uNr7INV3wPLzYR65cjhkN9FjIAndcnp2hsr+gE+QLwPLxfEK4X5s+UT7VrYBaB2q7Npu
8k6TsvgSG+kX09vtm/ww4rQiSMbZ4dx64EBag97QhuYeSODOGVTNog5z+HdmwhN6R12WnOkQJGmNNJ9ZzwKBr1v5
wyL6iOpumfrIV9YduI2bE8jQ69epGXmXQcVSpGtEpwvp7UqoxnSW7ycgiL2WjZfrYD/g0ngWexZud+ZfXRXdWox
Lut8F0TaKMS254wFPa9RzyTSv42PRcrkM9pY4MQGBpBZVMZ+oiCp3XYFwUd8yTv6oL8l50euDdfI78Psw+sLbN4m
dfktWpdeqovAks2yIkI4os8Hi2S/2DQK/LqgcJA8aUJwcSezsdVe8yULaicOZYx7Lje0HJyHX0WzqtNVs2S14RyG
JkYeTicMisYHbJaNGaIs06hgEv4Jj1xsWdynp/XWa1aXkQzBUwcQWajjChcOyTlIi/qMEqrQbwHyp7g0Vq5MM382
kgoP5UwhX3n3njwXpQLjLP6ItIMA1VtpcRZBMDiJddvgSaxhHIVsW31enmULKJ5KUFdV25wxS4ySf4iouupLzxLf
64JEH1GQZdPK/gwU6Q== greenlight@greenlight.alexedwards.net
```

And if you run the `ssh-add -l` command, you should see your new SSH key listed in the output, similar to this:

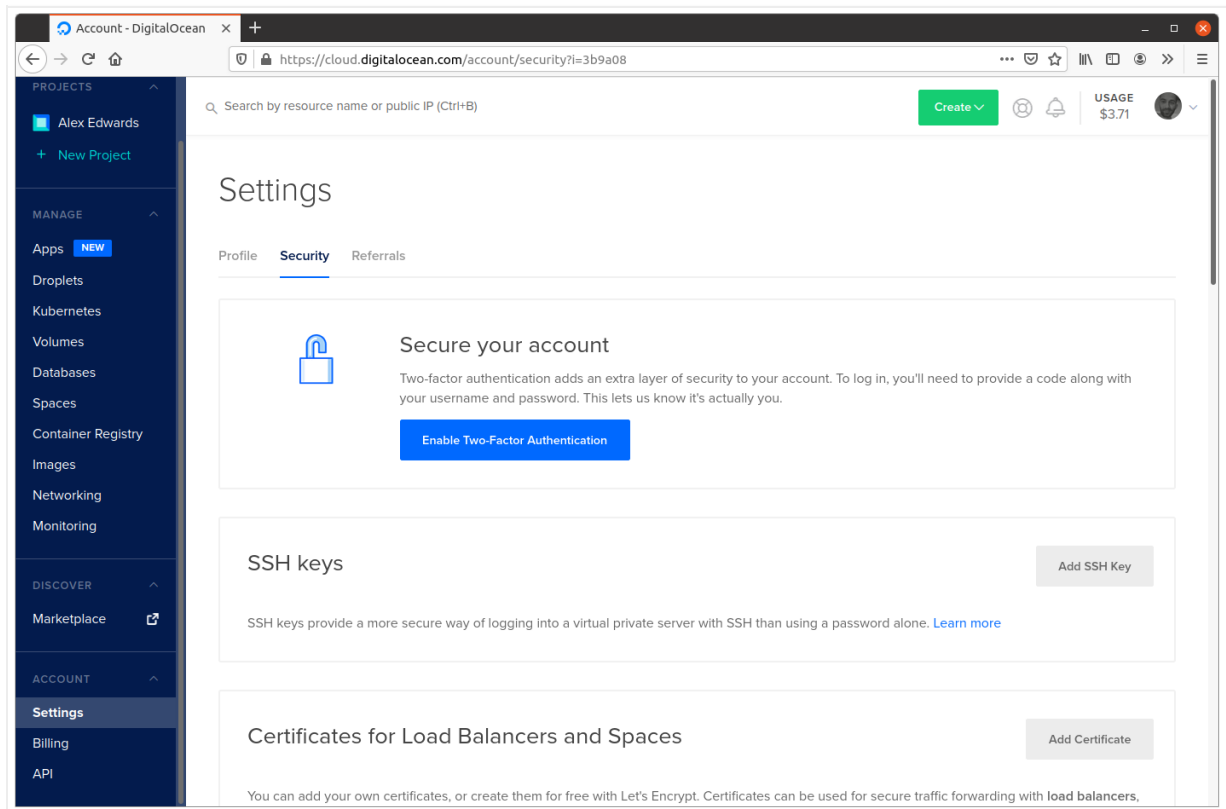
```
$ ssh-add -l
4096 SHA256:/ii7Wo90BdsoaMU1UAajmd/5vBSeVmbRtf7Qtoww8YM greenlight@greenlight.alexedwards.net (RSA)
```

If you don't see your key listed, then please add it to your *SSH agent* like so:

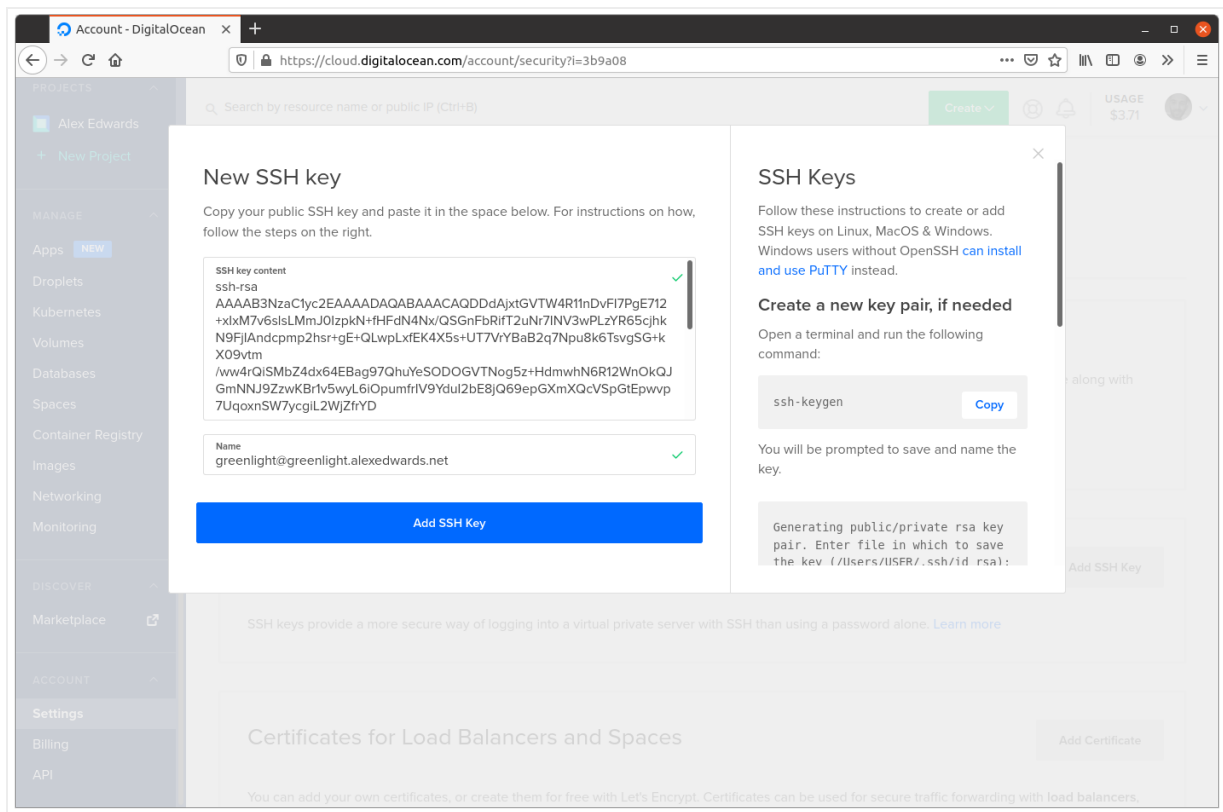

```
$ ssh-add $HOME/.ssh/id_rsa_greenlight
Enter passphrase for /home/alex/.ssh/id_rsa_greenlight:
Identity added: /home/alex/.ssh/id_rsa_greenlight (greenlight@greenlight.alexedwards.net)
```

Adding the SSH key to Digital Ocean

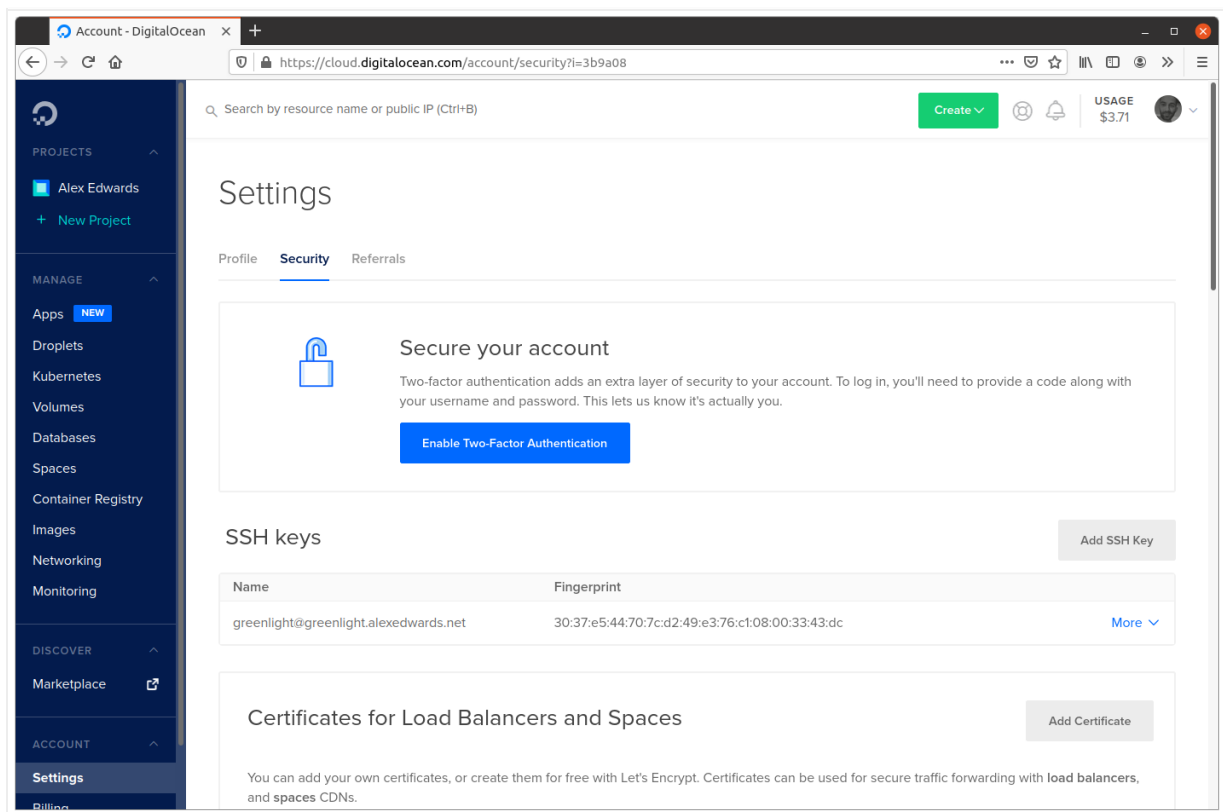
Now you've got a SSH keypair you can use, head back to your Digital Ocean control panel and navigate to the **Account › Settings › Security** screen.



Click the **Add SSH Key** button, then in the popup window that appears paste in the text contents from your `$HOME/.ssh/id_rsa_greenlight.pub` public key file, give it a memorable name, and submit the form, similar to the screenshot below.



The screen should then update to confirm that your SSH key has been successfully added, like so:



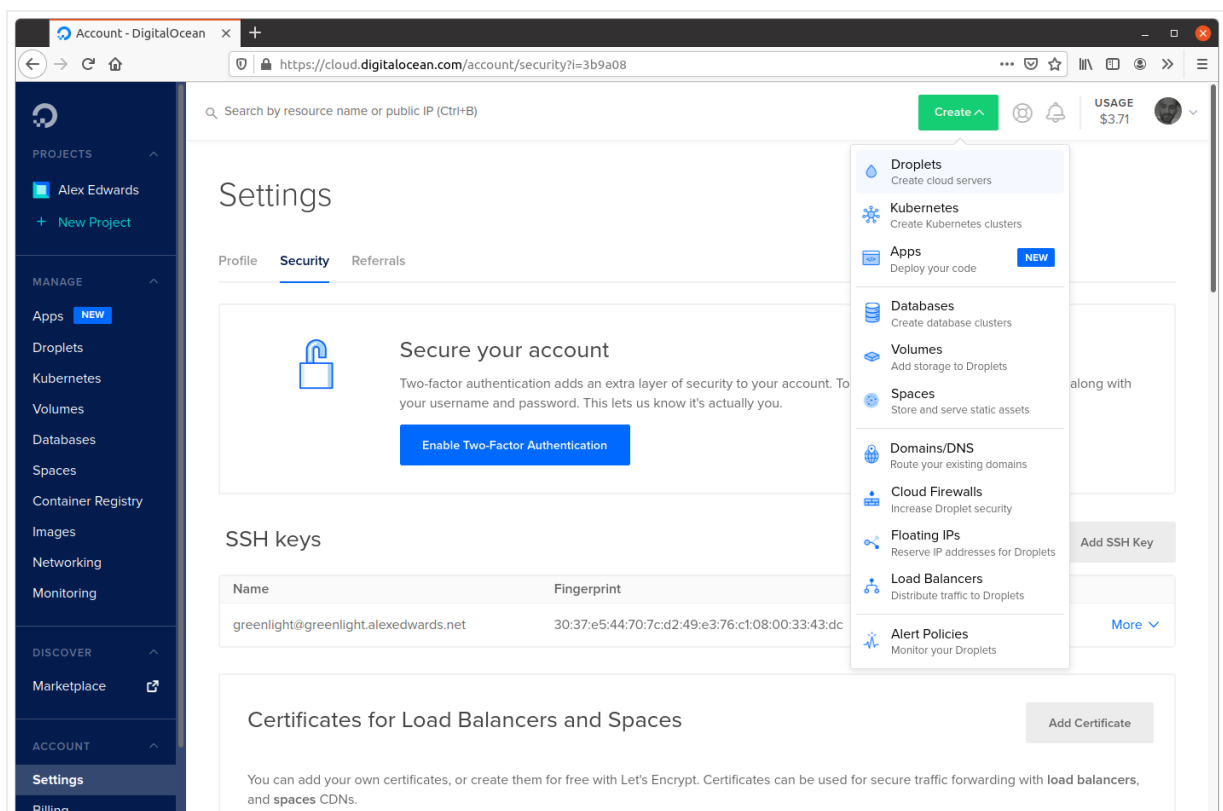
Creating a droplet

Now that you have a valid SSH key added to your account, it's time to actually create a droplet.

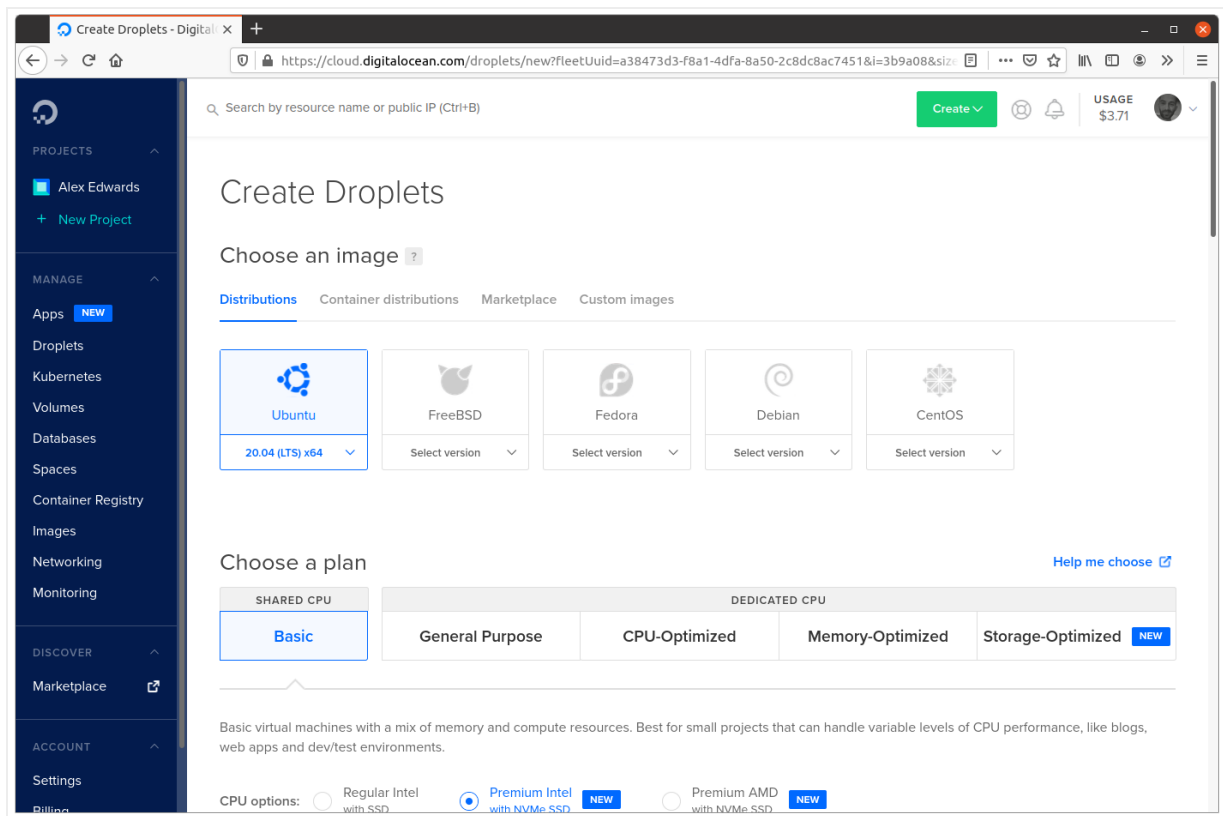
There are a couple of ways that you can do this. It's possible to do so programmatically via the [Digital Ocean API](#) or using the official [command-line tool](#), and if you need to create or manage a lot of servers then I recommend using these.

Or alternatively, it's possible to create a droplet manually via your control panel on the Digital Ocean website. This is the approach we'll take in this book, partly because it's simple enough to do as a one-off, and partly because it helps give overview of the available droplet settings if you haven't used Digital Ocean before.

Go ahead and click the green **Create** button in the top right corner and select **Droplets** from the dropdown menu:



This will then take you to the options page for creating a new droplet. The first thing to choose is the operating system for your droplet. If you're following along, please select **Ubuntu 20.04 (LTS) x64**.



The next step is to choose a plan which matches the technical specifications you need for the droplet. In this case we'll select the **Basic Regular Intel with SSD** plan at **\$5/month**, which will give us a virtual machine with 1GB RAM, 25GB of disk space and 1000GB of [outbound data transfer](#) each month (inbound data transfer is unrestricted).

Choose a plan

SHARED CPU

DEDICATED CPU

Basic

General Purpose

CPU-Optimized

Memory-Optimized

Storage-Optimized **NEW**

Basic virtual machines with a mix of memory and compute resources. Best for small projects that can handle variable levels of CPU performance, like blogs, web apps and dev/test environments.

CPU options: ☒ Regular Intel with SSD ☐ Premium Intel with NVMe SSD **NEW** ☐ Premium AMD with NVMe SSD **NEW**

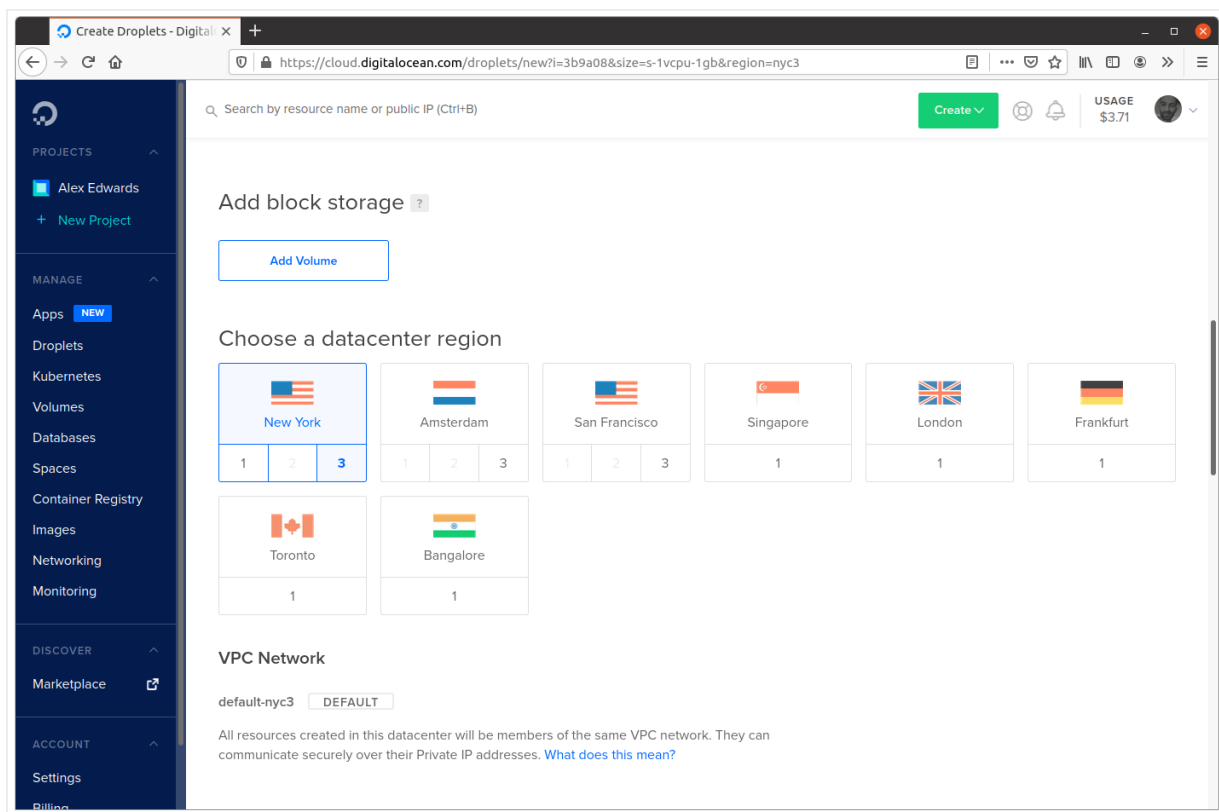
Plan	Price	RAM / CPU	SSD Disk	Transfer
Basic	\$5/mo \$0.007/hour	1 GB / 1 CPU	25 GB SSD Disk	1000 GB transfer
General Purpose	\$10/mo \$0.015/hour	2 GB / 1 CPU	50 GB SSD Disk	2 TB transfer
CPU-Optimized	\$15/mo \$0.022/hour	2 GB / 2 CPUs	60 GB SSD Disk	3 TB transfer
Memory-Optimized	\$20/mo \$0.030/hour	4 GB / 2 CPUs	80 GB SSD Disk	4 TB transfer
Memory-Optimized	\$40/mo \$0.060/hour	8 GB / 4 CPUs	160 GB SSD Disk	5 TB transfer
Storage-Optimized	\$80/mo \$0.119/hour	16 GB / 8 CPUs	320 GB SSD Disk	6 TB transfer

Our Basic Droplet plans, formerly called Standard Droplet plans, range from 1 GB of RAM to 16 GB of RAM. [General Purpose Droplets](#) have more overall resources and are best for production environment, and [Memory-Optimized Droplets](#) have more RAM and disk options for RAM intensive applications.

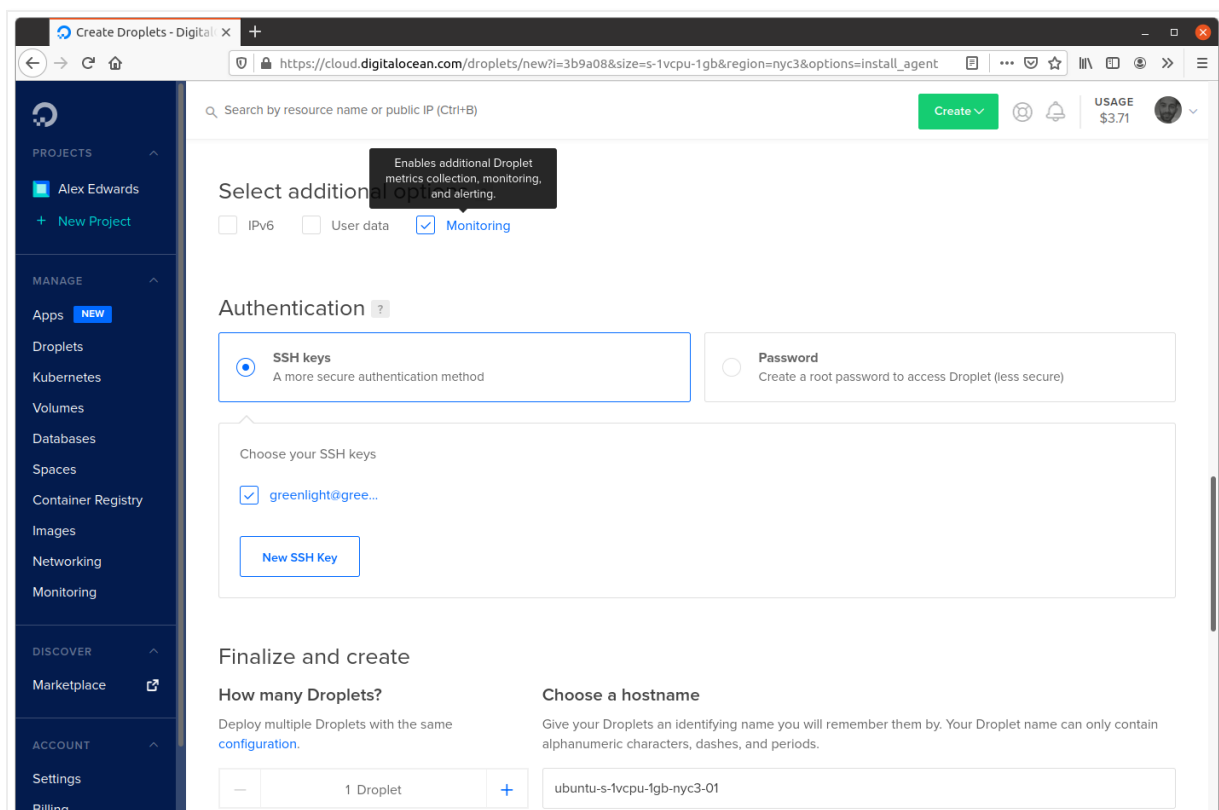
Each Droplet plan includes **free outbound data transfer** which is shared between all Droplets each billing cycle. Inbound bandwidth to Droplets is always free. [Learn more](#) or [try our price calculator](#).

The next option allows us to add [block storage](#) to the droplet. This is essentially a droplet-independent storage volume which acts like a local hard disk and can be moved between different droplets. It's not something we need right now, so you can skip this section.

After that we need to select the data center where our droplet will be physically hosted. I'm going to choose **New York 3** but feel free to pick an alternative location if you like.

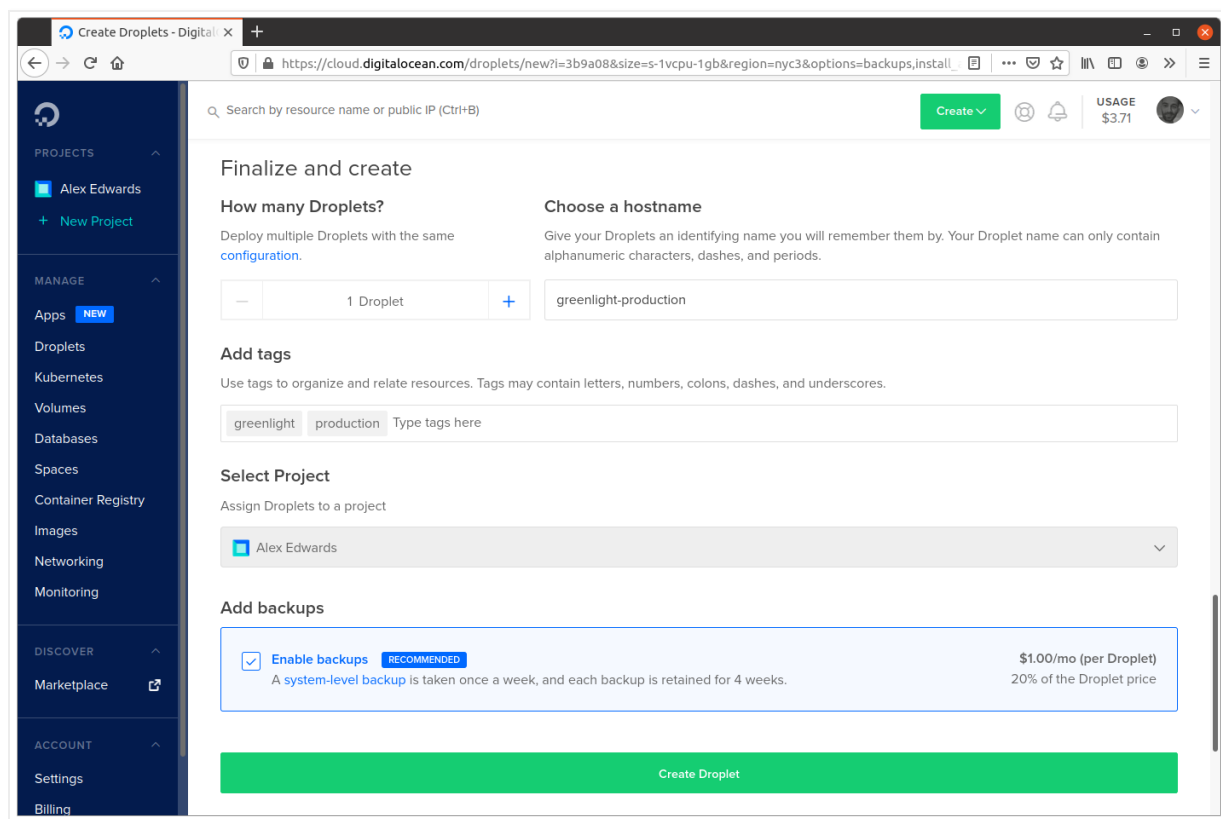


After that we can select some free “add-on” features for our droplet. In our case we’ll select **Monitoring**, which will allow you to later see graphs of various droplet statistics (like CPU, memory and disk use) in your Digital Ocean control panel — and you can also set up alerts if resource usage exceeds a certain threshold.



Under the **Authentication** section, make sure that **SSH keys** is selected as the authentication method and that the SSH key that you just uploaded is checked.

Then we get to the final configuration options.



The screenshot shows the DigitalOcean 'Finalize and create' page for a new Droplet. The left sidebar contains navigation links for PROJECTS, MANAGE (Apps, Droplets, Kubernetes, Volumes, Databases, Spaces, Container Registry, Images, Networking, Monitoring), DISCOVER (Marketplace), and ACCOUNT (Settings, Billing). The main content area is titled 'Finalize and create' and includes the following sections:

- How many Droplets?**: A section with a minus button, '1 Droplet', and a plus button. Below it, a note says 'Deploy multiple Droplets with the same configuration.'
- Choose a hostname**: A text input field containing 'greenlight-production'. A note below says 'Give your Droplets an identifying name you will remember them by. Your Droplet name can only contain alphanumeric characters, dashes, and periods.'
- Add tags**: A section with a note 'Use tags to organize and relate resources. Tags may contain letters, numbers, colons, dashes, and underscores.' Below it, two tags 'greenlight' and 'production' are shown, followed by a text input field 'Type tags here'.
- Select Project**: A section with a note 'Assign Droplets to a project'. Below it, a dropdown menu shows 'Alex Edwards' as the selected project.
- Add backups**: A section with a checkbox 'Enable backups' (checked) and a 'RECOMMENDED' badge. Below it, a note says 'A system-level backup is taken once a week, and each backup is retained for 4 weeks.' To the right, the cost is listed as '\$1.00/mo (per Droplet)' and '20% of the Droplet price'.

At the bottom of the main content area is a large green button labeled 'Create Droplet'.

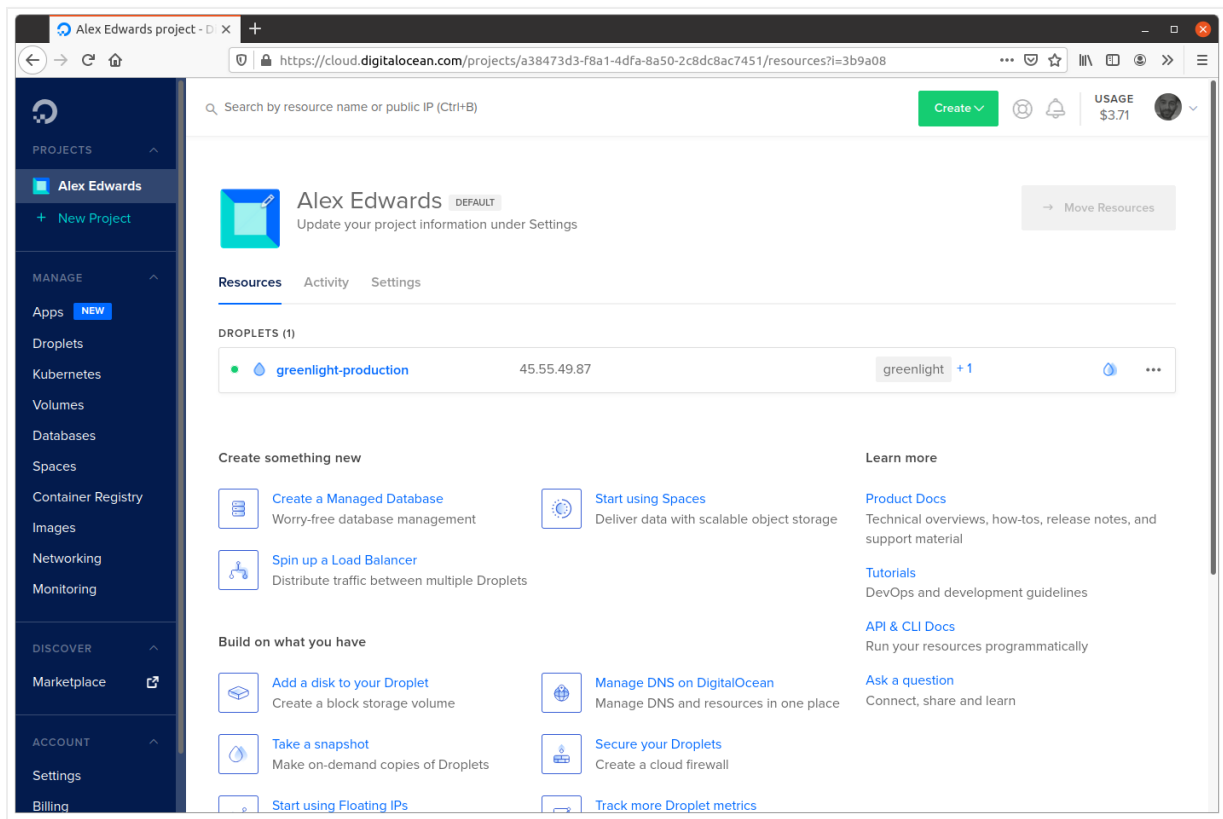
We only need **one droplet** right now, so you can leave that as the default.

You should also add a **hostname** for the droplet. Amongst other things, the hostname is used as the main identifier for the droplet in the Digital Ocean control panel, and it's also what you'll see when you SSH into the droplet later to administer it. So you should pick a name that is sensible and easy-to-recognize. I'm going to use the hostname **greenlight-production**, but feel free to use something different if you like.

Adding **tags** to your droplet is completely optional, but if you do a lot of work with Digital Ocean they can be a useful way to help filter and manage droplets. I'll use the tags **greenlight** and **production** here.

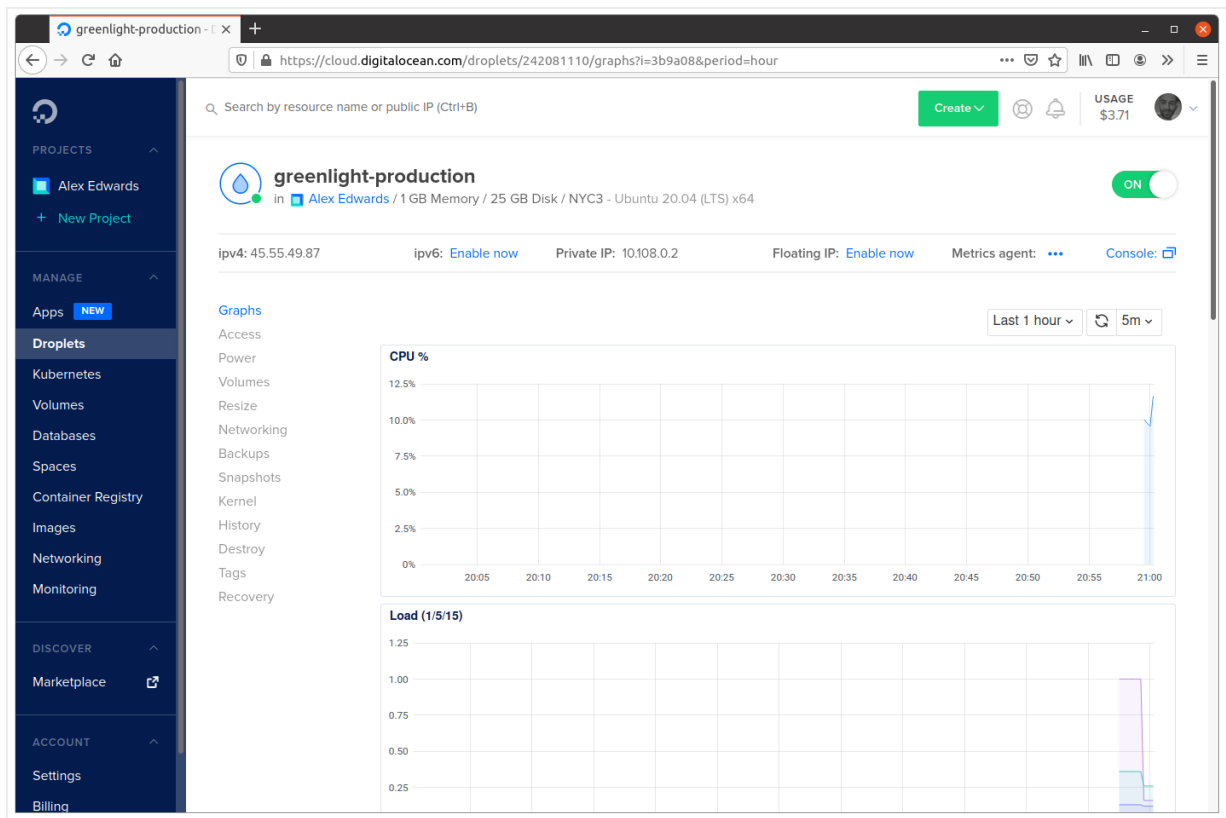
Then lastly you can choose to pay an additional 20% charge to enable automated **droplet backups**. If you select this option, then Digital Ocean will take a 'snapshot' of your droplet once a week and store it for a period of 4 weeks. You can then restore a droplet to its snapshotted state via the control panel if you ever need to. It's entirely up to you whether to enable backups or not — but it's a simple and hassle-free safety net.

Once that's all set, go ahead and click the **Create Droplet** button at the foot of the screen. You should see a progress bar while the droplet is being set up for you, and after a minute or two the droplet should be live and ready to use.



The most important thing at this point is noting the IP address for the droplet, which in my case is **45.55.49.87**.

If you like, you can also click on the droplet hostname to see more detailed information about the droplet (including the monitoring statistics) and make any further configuration and management changes if you need to.



OK, now that the droplet is set up, it's time for the moment of truth!

Open a new terminal window and try connecting to the droplet via SSH as the **root** user, using the droplet IP address. Like so...

```
$ ssh root@45.55.49.87
The authenticity of host '45.55.49.87 (45.55.49.87)' can't be established.
ECDSA key fingerprint is SHA256:HWGdr4i2xF0yoU3GLRcVOYV/pqJ45pwLXhKWAjq4ahw.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '45.55.49.87' (ECDSA) to the list of known hosts.
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-51-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

System information as of Sun Apr 18 19:02:59 UTC 2021

System load: 0.0           Users logged in: 0
Usage of /:  5.9% of 24.06GB IPv4 address for eth0: 45.55.49.87
Memory usage: 21%         IPv4 address for eth0: 10.17.0.5
Swap usage:  0%           IPv4 address for eth1: 10.108.0.2
Processes:   103

126 updates can be installed immediately.
60 of these updates are security updates.
To see these additional updates run: apt list --upgradable

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

root@greenlight-production:~#
```

Note: As this is the first time that you're using SSH to connect to this droplet you'll get a message saying that "the authenticity of host can't be established". This is [fine in this instance](#) and just type **yes** to continue. If your SSH key is protected with a passphrase, you may also be prompted to enter that too.

Great, that looks like it's working well. Our Ubuntu Linux droplet is up and running, and we've been able to successfully connect to it as the **root** user over SSH.

You can go ahead and type **exit** to terminate the SSH connection and return to the terminal on your local machine, like so:

```
root@greenlight-production:~# exit
logout
Connection to 45.55.49.87 closed.
```

Server Configuration and Installing Software

Now that our Ubuntu Linux droplet has been successfully commissioned, we need to do some housekeeping to secure the server and get it ready-to-use. Rather than do this manually, in this chapter we're going to create a reusable script to automate these setup tasks.

Note: In this chapter we'll use a regular bash script to perform the setup tasks, but if you're happy using a configuration management tool such as [Ansible](#), then please feel free to implement the same setup process using that instead.

At a high level we want our setup script to do the following things:

- Update all packages on the server, including applying security updates.
- Set the server timezone (in my case I'll set it to `America/New_York`) and install support for all [locales](#).
- Create a `greenlight` user on the server, which we can use for day-to-day maintenance and for running our API application (rather than using the all-powerful `root` user account). We should also add the `greenlight` user to the `sudo` group, so that it can perform actions as `root` if necessary.
- Copy the `root` user's `$HOME/.ssh` directory into the `greenlight` users home directory. This will enable us to authenticate as the `greenlight` user using the same SSH key pair that we used to authenticate as the `root` user. We should also force the `greenlight` user to set a new password the first time they log in.
- Configure firewall settings to only permit traffic on ports `22` (SSH), `80` (HTTP) and `443` (HTTPS). We'll also install [fail2ban](#) to automatically temporarily ban an IP address if it makes too many failed SSH login attempts.
- Install PostgreSQL. We'll also create the `greenlight` database and user, and create a system-wide `GREENLIGHT_DB_DSN` environment variable for connecting to the database.
- Install the `migrate` tool, using the [pre-built binaries](#) from GitHub, in exactly the same

way that we did earlier in the book.

- Install Caddy by following the [official installation instructions](#) for Ubuntu.
- Reboot the droplet.

If you're following along, create a new `remote/setup` folder in your project directory and add a script file called `01.sh`, like so:

```
$ mkdir -p remote/setup
$ touch remote/setup/01.sh
```

Then add the following code:

File: remote/setup/01.sh

```
#!/bin/bash
set -eu

# ===== #
# VARIABLES
# ===== #

# Set the timezone for the server. A full list of available timezones can be found by
# running timedatectl list-timezones.
TIMEZONE=America/New_York

# Set the name of the new user to create.
USERNAME=greenlight

# Prompt to enter a password for the PostgreSQL greenlight user (rather than hard-coding
# a password in this script).
read -p "Enter password for greenlight DB user: " DB_PASSWORD

# Force all output to be presented in en_US for the duration of this script. This avoids
# any "setting locale failed" errors while this script is running, before we have
# installed support for all locales. Do not change this setting!
export LC_ALL=en_US.UTF-8

# ===== #
# SCRIPT LOGIC
# ===== #

# Enable the "universe" repository.
add-apt-repository --yes universe

# Update all software packages. Using the --force-confnew flag means that configuration
# files will be replaced if newer ones are available.
apt update
apt --yes -o Dpkg::Options::="--force-confnew" upgrade

# Set the system timezone and install all locales.
timedatectl set-timezone ${TIMEZONE}
apt --yes install locales-all

# Add the new user (and give them sudo privileges).
useradd --create-home --shell "/bin/bash" --groups sudo "${USERNAME}"
```

```

# Force a password to be set for the new user the first time they log in.
passwd --delete "${USERNAME}"
chage --lastday 0 "${USERNAME}"

# Copy the SSH keys from the root user to the new user.
rsync --archive --chown=${USERNAME}:${USERNAME} /root/.ssh /home/${USERNAME}

# Configure the firewall to allow SSH, HTTP and HTTPS traffic.
ufw allow 22
ufw allow 80/tcp
ufw allow 443/tcp
ufw --force enable

# Install fail2ban.
apt --yes install fail2ban

# Install the migrate CLI tool.
curl -L https://github.com/golang-migrate/migrate/releases/download/v4.14.1/migrate.linux-amd64.tar.gz | tar xzv
mv migrate.linux-amd64 /usr/local/bin/migrate

# Install PostgreSQL.
apt --yes install postgresql

# Set up the greenlight DB and create a user account with the password entered earlier.
sudo -i -u postgres psql -c "CREATE DATABASE greenlight"
sudo -i -u postgres psql -d greenlight -c "CREATE EXTENSION IF NOT EXISTS citext"
sudo -i -u postgres psql -d greenlight -c "CREATE ROLE greenlight WITH LOGIN PASSWORD '${DB_PASSWORD}'"

# Add a DSN for connecting to the greenlight database to the system-wide environment
# variables in the /etc/environment file.
echo "GREENLIGHT_DB_DSN='postgres://greenlight:${DB_PASSWORD}@localhost/greenlight'" >> /etc/environment

# Install Caddy (see https://caddyserver.com/docs/install#debian-ubuntu-raspbian).
apt --yes install -y debian-keyring debian-archive-keyring apt-transport-https
curl -L https://dl.cloudsmith.io/public/caddy/stable/gpg.key | sudo apt-key add -
curl -L https://dl.cloudsmith.io/public/caddy/stable/debian.deb.txt | sudo tee -a /etc/apt/sources.list.d/caddy-stable.list
apt update
apt --yes install caddy

echo "Script complete! Rebooting..."
reboot

```

OK, let's now run this script on our new Digital Ocean droplet. This will be a two-step process:

1. First we need to copy the script to the droplet (which we will do using `rsync`).
2. Then we need to connect to the droplet via SSH and execute the script.

Go ahead and run the following command to `rsync` the contents of the `/remote/setup` folder to the `root` user's home directory on the droplet. Remember to replace the IP address with your own!

```

$ rsync -rP --delete ./remote/setup root@45.55.49.87:/root
sending incremental file list
setup/
setup/01.sh
      3,327 100%   0.00kB/s   0:00:00 (xfr#1, to-chk=0/2)

```

Note: In this `rsync` command the `-r` flag indicates that we want to copy the contents of `./remote/setup` recursively, the `-P` flag indicates that we want to display progress of the transfer, and the `--delete` flag indicates that we want to delete any extraneous files from destination directory on the droplet.

Now that a copy of our setup script is on the droplet, let's use the `ssh` command to execute the script on the remote machine as the `root` user. We'll use the `-t` flag to force *pseudo-terminal allocation*, which is useful when executing screen-based programs on a remote machine.

Go ahead and run the script, entering a password for the `greenlight PostgreSQL user`, like so:

```
$ ssh -t root@45.55.49.87 "bash /root/setup/01.sh"
Enter password for greenlight DB user: pa55word1234
'universe' distribution component enabled for all sources.
Hit:1 http://security.ubuntu.com/ubuntu focal-security InRelease
Hit:2 https://repos.insights.digitalocean.com/apt/do-agent main InRelease
Get:3 http://mirrors.digitalocean.com/ubuntu focal InRelease [265 kB]
Hit:4 http://mirrors.digitalocean.com/ubuntu focal-updates InRelease
Hit:5 http://mirrors.digitalocean.com/ubuntu focal-backports InRelease
...
Script complete! Rebooting...
Connection to 45.55.49.87 closed by remote host.
Connection to 45.55.49.87 closed.
```

Important: While this script is running you will get a message saying: A new version of configuration file `/etc/ssh/sshd_config` is available, but the version installed currently has been locally modified.

This is because Digital Ocean automatically makes some adaptations to the `sshd_config` file, including disabling password-based access when you are using SSH keys for authentication. Please select `keep the local version currently installed` and then `<Ok>` to continue.

After a few minutes the script should complete successfully and the droplet will be rebooted — which will also kick you off the SSH connection.

Connecting as the greenlight user

After waiting a minute for the reboot to complete, try connecting to the droplet as the

greenlight user over SSH. This should work correctly (and the SSH key pair you created in the previous chapter should be used to authenticate the connection) *but you will be prompted to set a password.*

```
$ ssh greenlight@45.55.49.87
You are required to change your password immediately (administrator enforced)
Welcome to Ubuntu 20.04.2 LTS (GNU/Linux 5.4.0-72-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

System information as of Sun Apr 18 15:51:33 EDT 2021

System load:  0.0               Users logged in:      0
Usage of /:   9.2% of 24.06GB   IPv4 address for eth0: 45.55.49.87
Memory usage: 20%              IPv4 address for eth0: 10.17.0.5
Swap usage:   0%               IPv4 address for eth1: 10.108.0.2
Processes:   109

0 updates can be installed immediately.
0 of these updates are security updates.

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

WARNING: Your password has expired.
You must change your password now and login again!
New password:
Retype new password:
passwd: password updated successfully
Connection to 45.55.49.87 closed.
```

The password you enter here will be the regular password for the **greenlight** user on your droplet (i.e. the password you will need to type whenever you are logged in and want to execute a **sudo** command). Just to be clear, it is *not* a SSH password for connecting as the **greenlight** user.

Go ahead and enter whatever password you would like, confirm it, and check that you see a message saying that the password change has been successful. The SSH connection will then be automatically terminated.

If you re-connect as the **greenlight** user, everything should now work normally and you should be able to execute commands on the droplet. Like so:

```
$ ssh greenlight@45.55.49.87
Welcome to Ubuntu 20.04.2 LTS (GNU/Linux 5.4.0-65-generic x86_64)

...

greenlight@greenlight-production:~$ whoami
greenlight
```

While we're connected to the droplet, let's take a quick look around and check that some key things are working as we would expect.

First let's check that the **migrate** binary has been downloaded and is on the system path by running it with the **-version** flag. All being well, this should print the **migrate** version number like so:

```
greenlight@greenlight-production:~$ migrate -version
4.14.1
```

Next let's verify that PostgreSQL is running, and that we can connect to it using the DSN in the **GREENLIGHT_DB_DSN** environment variable:

```
greenlight@greenlight-production:~$ psql $GREENLIGHT_DB_DSN
psql (12.6 (Ubuntu 12.6-0ubuntu0.20.04.1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression: off)
Type "help" for help.

greenlight=> exit
```

Finally, let's check that Caddy is running by calling **systemctl status caddy** to see the status of the Caddy background service. You'll need to use **sudo** here to run this command as **root**.

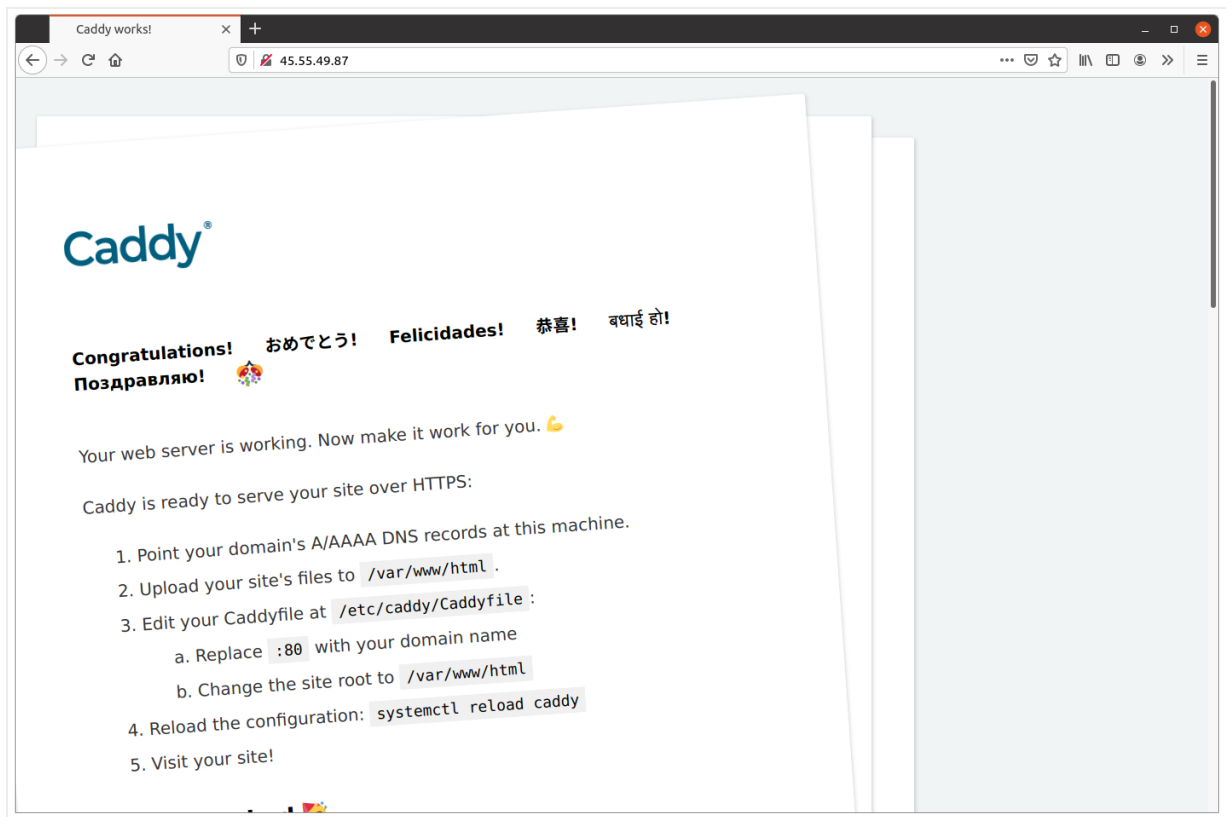
```
greenlight@greenlight-production:~$ sudo systemctl status caddy
● caddy.service - Caddy
   Loaded: loaded (/lib/systemd/system/caddy.service; enabled; vendor preset: enabled)
   Active: active (running) since Sun 2021-04-18 15:23:32 EDT; 37min ago
     Docs: https://caddyserver.com/docs/
   Main PID: 586 (caddy)
    Tasks: 7 (limit: 1136)
   Memory: 37.6M
   CGroup: /system.slice/caddy.service
           └─586 /usr/bin/caddy run --environ --config /etc/caddy/Caddyfile

Apr 18 15:23:33 greenlight-production caddy[586]: USER=caddy
Apr 18 15:23:33 greenlight-production caddy[586]: INVOCATION_ID=2e2dc35c3cf44786b7103847fd9a9ad8
...
```

You should see that the status of the Caddy service is **active (running)** — confirming that

Caddy is working successfully.

This means that you should be able to visit your droplet directly in a web browser via `http://<your_droplet_ip>`, and you should see the following Caddy welcome page.



Connecting to the droplet

To make connecting to the droplet a bit easier, and so we don't have to remember the IP address, let's quickly add a makefile rule for initializing a SSH connection to the droplet as the `greenlight` user. Like so:

File: Makefile

```
...

# ===== #
# PRODUCTION
# ===== #

production_host_ip = '45.55.49.87'

## production/connect: connect to the production server
.PHONY: production/connect
production/connect:
    ssh greenlight@${production_host_ip}
```

Once that's in place, you can then connect to your droplet whenever you need to by simply typing `make production/connect`:

```
$ make production/connect
ssh greenlight@45.55.49.87
Welcome to Ubuntu 20.04.2 LTS (GNU/Linux 5.4.0-65-generic x86_64)

...

greenlight@greenlight-production:~$
```

Additional Information

Future changes to the droplet configuration

If you need to make further changes to your droplet configuration or settings, you can create an additional `remote/setup/02.sh` script and then execute it in the following way:

```
$ rsync -rP --delete ./remote/setup greenlight@45.55.49.87:~
$ ssh -t greenlight@45.55.49.87 "sudo bash /home/greenlight/setup/02.sh"
```

Deployment and Executing Migrations

At this point our droplet is set up with all the software and user accounts that we need, so let's move on to the process of deploying and running our API application.

At a very high-level, our deployment process will consist of three actions:

1. Copying the application binary and SQL migration files to the droplet.
2. Executing the migrations against the PostgreSQL database on the droplet.
3. Starting the application binary as a *background service*.

For now we'll just focus on steps 1 and 2, and tackle running our application as a *background service* in the next chapter.

Let's begin by creating a new `make production/deploy/api` rule in our makefile, which we will use to execute these first two steps automatically. Like so:

File: Makefile

```
...

# ===== #
# PRODUCTION
# ===== #

production_host_ip = "45.55.49.87"

## production/connect: connect to the production server
.PHONY: production/connect
production/connect:
    ssh greenlight@${production_host_ip}

## production/deploy/api: deploy the api to production
.PHONY: production/deploy/api
production/deploy/api:
    rsync -P ./bin/linux_amd64/api greenlight@${production_host_ip}:~
    rsync -rP --delete ./migrations greenlight@${production_host_ip}:~
    ssh -t greenlight@${production_host_ip} 'migrate -path ~/migrations -database $$GREENLIGHT_DB_DSN up'
```

Let's quickly break down what this new rule is doing.

First, it uses the `rsync` command to copy the `./bin/linux_amd64/api` executable binary (the one specifically built for Linux) and the `./migrations` folder into the home directory for the `greenlight` user on the droplet.

Then it uses the `ssh` command with the `-t` flag to run these database migrations on our

droplet as the **greenlight** user with the following command:

```
'migrate -path ~/migrations -database $$GREENLIGHT_DB_DSN up'
```

Because the **\$** character has a special meaning in makefiles, we are *escaping it* in the command above by prefixing it with an **additional dollar character** like **\$\$**. This means that the command that actually runs on our droplet will be

'migrate -path ~/migrations -database \$GREENLIGHT_DB_DSN up' , and in turn, the migration will be run *using the droplet environment variable* **GREENLIGHT_DB_DSN**.

It's also important to note that we're surrounding this command with single quotes. If we used double quotes, it would be an *interpreted string* and we would need to use an additional escape character **** like so:

```
"migrate -path ~/migrations -database \"$$GREENLIGHT_DB_DSN up"
```

Alright, let's try this out!

Go ahead and execute the **make production/deploy/api** rule that we just made. You should see that all the files copy across successfully (this may take a minute or two to complete depending on your connection speed), and the migrations are then applied to your database. Like so:

```
$ make production/deploy/api
rsync -rP --delete ./bin/linux_amd64/api greenlight@"45.55.49.87":~
api
 7,618,560 100% 119.34kB/s   0:01:02 (xfr#1, to-chk=13/14)
rsync -rP --delete ./migrations greenlight@"45.55.49.87":~
sending incremental file list
migrations/
migrations/000001_create_movies_table.down.sql
 28 100% 27.34kB/s   0:00:00 (xfr#2, to-chk=11/14)
migrations/000001_create_movies_table.up.sql
 286 100% 279.30kB/s   0:00:00 (xfr#3, to-chk=10/14)
migrations/000002_add_movies_check_constraints.down.sql
 198 100% 193.36kB/s   0:00:00 (xfr#4, to-chk=9/14)
migrations/000002_add_movies_check_constraints.up.sql
 289 100% 282.23kB/s   0:00:00 (xfr#5, to-chk=8/14)
migrations/000003_add_movies_indexes.down.sql
 78 100% 76.17kB/s   0:00:00 (xfr#6, to-chk=7/14)
migrations/000003_add_movies_indexes.up.sql
 170 100% 166.02kB/s   0:00:00 (xfr#7, to-chk=6/14)
migrations/000004_create_users_table.down.sql
 27 100% 26.37kB/s   0:00:00 (xfr#8, to-chk=5/14)
migrations/000004_create_users_table.up.sql
 294 100% 287.11kB/s   0:00:00 (xfr#9, to-chk=4/14)
migrations/000005_create_tokens_table.down.sql
 28 100% 27.34kB/s   0:00:00 (xfr#10, to-chk=3/14)
migrations/000005_create_tokens_table.up.sql
 203 100% 99.12kB/s   0:00:00 (xfr#11, to-chk=2/14)
migrations/000006_add_permissions.down.sql
 73 100% 35.64kB/s   0:00:00 (xfr#12, to-chk=1/14)
migrations/000006_add_permissions.up.sql
 452 100% 220.70kB/s   0:00:00 (xfr#13, to-chk=0/14)
ssh -t greenlight@"45.55.49.87" 'migrate -path ~/migrations -database $GREENLIGHT_DB_DSN up'
1/u create_movies_table (11.782733ms)
2/u add_movies_check_constraints (23.109006ms)
3/u add_movies_indexes (30.61223ms)
4/u create_users_table (39.890662ms)
5/u create_tokens_table (48.659641ms)
6/u add_permissions (58.23243ms)
Connection to 45.55.49.87 closed.
```

Let's quickly connect to our droplet and verify that everything has worked.

```
$ make production/connect
ssh greenlight@'45.55.49.87'
Welcome to Ubuntu 20.04.2 LTS (GNU/Linux 5.4.0-65-generic x86_64)

...
greenlight@greenlight-production:~$ ls -R
.:
api migrations

./migrations:
000001_create_movies_table.down.sql
000001_create_movies_table.up.sql
000002_add_movies_check_constraints.down.sql
000002_add_movies_check_constraints.up.sql
000003_add_movies_indexes.down.sql
000003_add_movies_indexes.up.sql
000004_create_users_table.down.sql
000004_create_users_table.up.sql
000005_create_tokens_table.down.sql
000005_create_tokens_table.up.sql
000006_add_permissions.down.sql
000006_add_permissions.up.sql
```

So far so good. We can see that the home directory of our **greenlight** user contains the **api** executable binary and a folder containing the migration files.

Let's also connect to the database using **psql** and verify that the tables have been created by using the **\dt** meta command. Your output should look similar to this:

```
greenlight@greenlight-production:~$ psql $GREENLIGHT_DB_DSN
psql (12.6 (Ubuntu 12.6-0ubuntu0.20.04.1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression: off)
Type "help" for help.

greenlight=> \dt
          List of relations
Schema |      Name      | Type | Owner
-----+-----+-----+-----
public | movies          | table | greenlight
public | permissions     | table | greenlight
public | schema_migrations | table | greenlight
public | tokens          | table | greenlight
public | users           | table | greenlight
public | users_permissions | table | greenlight
(6 rows)
```

Running the API

While we're connected to the droplet, let's try running the **api** executable binary. We can't listen for incoming connections on port **80** because Caddy is already using this, so we'll listen on the unrestricted port **4000** instead.

If you've been following along, port **4000** on your droplet should currently be blocked by the

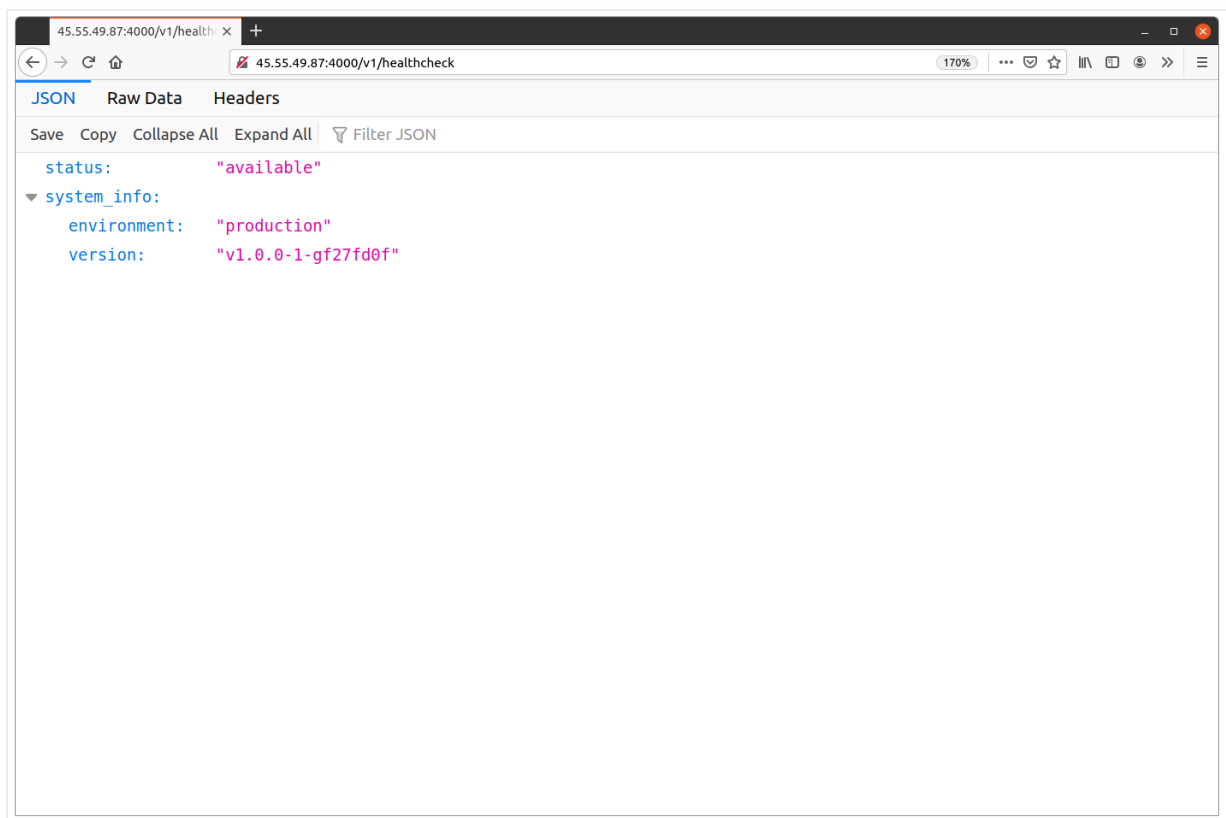
firewall rules, so we'll need to relax this temporarily to allow incoming requests. Go ahead and do that like so:

```
greenlight@greenlight-production:~$ sudo ufw allow 4000/tcp
Rule added
Rule added (v6)
```

And then start the API with the following command:

```
greenlight@greenlight-production:~$ ./api -port=4000 -db-dsn=$GREENLIGHT_DB_DSN -env=production
{"level":"INFO","time":"2021-04-19T07:10:55Z","message":"database connection pool established"}
{"level":"INFO","time":"2021-04-19T07:10:55Z","message":"starting server","properties":{"addr":":4000","env":"production"}}
```

At this point you should be able to visit http://<your_droplet_ip>:4000/v1/healthcheck in your web browser and get a successful response from the healthcheck endpoint, similar to this:



Lastly, head back to your SSH terminal window and press **Ctrl+C** to stop the API from running on your droplet. You should see it gracefully shutdown like so:

```
greenlight@greenlight-production:~$ /home/greenlight/api -port=4000 -db-dsn=$GREENLIGHT_DB_DSN -env=production
{"level":"INFO","time":"2021-04-19T07:10:55Z","message":"database connection pool established"}
{"level":"INFO","time":"2021-04-19T07:10:55Z","message":"starting server","properties":{"addr":":4000","env":"production"}}
^C{"level":"INFO","time":"2021-04-19T07:13:23Z","message":"caught signal","properties":{"signal":"interrupt"}}
{"level":"INFO","time":"2021-04-19T07:13:23Z","message":"completing background tasks","properties":{"addr":":4000"}}
{"level":"INFO","time":"2021-04-19T07:13:23Z","message":"stopped server","properties":{"addr":":4000"}}
```


Running the API as a Background Service

Now that we know our API executable works fine on our production droplet, the next step is to configure it to run as a *background service*, including starting up automatically when the droplet is rebooted.

There are a few different tools we could use to do this, but in this book we will use [systemd](#) — a collection of tools for managing services that ships with Ubuntu (and many other Linux distributions).

In order to run our API application as a background service, the first thing we need to do is make a [unit file](#), which informs systemd how and when to run the service.

If you're following along, head back to a terminal window on your local machine and create a new `remote/production/api.service` file in your project directory:

```
$ mkdir remote/production
$ touch remote/production/api.service
```

And then add the following markup:

File: remote/production/api.service

```
[Unit]
# Description is a human-readable name for the service.
Description=Greenlight API service

# Wait until PostgreSQL is running and the network is "up" before starting the service.
After=postgresql.service
After=network-online.target
Wants=network-online.target

# Configure service start rate limiting. If the service is (re)started more than 5 times
# in 600 seconds then don't permit it to start anymore.
StartLimitIntervalSec=600
StartLimitBurst=5

[Service]
# Execute the API binary as the greenlight user, loading the environment variables from
# /etc/environment and using the working directory /home/greenlight.
Type=exec
User=greenlight
Group=greenlight
EnvironmentFile=/etc/environment
WorkingDirectory=/home/greenlight
ExecStart=/home/greenlight/api -port=4000 -db-dsn=${GREENLIGHT_DB_DSN} -env=production

# Automatically restart the service after a 5-second wait if it exits with a non-zero
# exit code. If it restarts more than 5 times in 600 seconds, then the rate limit we
# configured above will be hit and it won't be restarted anymore.
Restart=on-failure
RestartSec=5

[Install]
# Start the service automatically at boot time (the 'multi-user.target' describes a boot
# state when the system will accept logins).
WantedBy=multi-user.target
```

Now that we've got a unit file set up, the next step is to install this unit file on our droplet and start up the service. Essentially we need to do three things:

1. To install the unit file, we need to copy it into the `/etc/systemd/system/` folder on our droplet. Because this folder is owned by the `root` user on our droplet, we need break this down into two steps: first we need to copy the unit file into the `greenlight` user's home directory, and secondly use the `sudo mv` command to move it to its final location.
2. Then we need to run the `systemctl enable api` command on our droplet to make `systemd` aware of the new unit file and automatically enable the service when the droplet is rebooted.
3. Finally, we need to run `systemctl restart api` to start (or restart) the service.

All three of these tasks will need to be run with `sudo`.

Let's update the `production/deploy/api` makefile rule to automate these tasks for us. Like so:

File: Makefile

```
...

# ===== #
# PRODUCTION
# ===== #

production_host_ip = '45.55.49.87'

...

## production/deploy/api: deploy the api to production
.PHONY: production/deploy/api
production/deploy/api:
  rsync -P ./bin/linux_amd64/api greenlight@${production_host_ip}:~
  rsync -rP --delete ./migrations greenlight@${production_host_ip}:~
  rsync -P ./remote/production/api.service greenlight@${production_host_ip}:~
  ssh -t greenlight@${production_host_ip} '\
    migrate -path ~/migrations -database $$GREENLIGHT_DB_DSN up \
    && sudo mv ~/api.service /etc/systemd/system/ \
    && sudo systemctl enable api \
    && sudo systemctl restart api \
  '
```

Save the makefile, then go ahead and run the `production/deploy/api` rule again. You should be prompted to enter the password for the `greenlight` user when it's executing, and the rule should complete without any errors. The output you see should look similar to this:

```

$ make production/deploy/api
rsync -P ./bin/linux_amd64/api greenlight@"45.55.49.87":~
api
  7,634,944 100%  70.67MB/s   0:00:00 (xfr#1, to-chk=0/1)
rsync -rP --delete ./migrations greenlight@"45.55.49.87":~
sending incremental file list
migrations/000001_create_movies_table.down.sql
  28 100%  0.00kB/s   0:00:00 (xfr#1, to-chk=11/13)
migrations/000001_create_movies_table.up.sql
 286 100% 279.30kB/s   0:00:00 (xfr#2, to-chk=10/13)
migrations/000002_add_movies_check_constraints.down.sql
 198 100% 193.36kB/s   0:00:00 (xfr#3, to-chk=9/13)
migrations/000002_add_movies_check_constraints.up.sql
 289 100% 282.23kB/s   0:00:00 (xfr#4, to-chk=8/13)
migrations/000003_add_movies_indexes.down.sql
  78 100%  76.17kB/s   0:00:00 (xfr#5, to-chk=7/13)
migrations/000003_add_movies_indexes.up.sql
 170 100% 166.02kB/s   0:00:00 (xfr#6, to-chk=6/13)
migrations/000004_create_users_table.down.sql
  27 100%  26.37kB/s   0:00:00 (xfr#7, to-chk=5/13)
migrations/000004_create_users_table.up.sql
 294 100% 287.11kB/s   0:00:00 (xfr#8, to-chk=4/13)
migrations/000005_create_tokens_table.down.sql
  28 100%  27.34kB/s   0:00:00 (xfr#9, to-chk=3/13)
migrations/000005_create_tokens_table.up.sql
 203 100% 198.24kB/s   0:00:00 (xfr#10, to-chk=2/13)
migrations/000006_add_permissions.down.sql
  73 100%  71.29kB/s   0:00:00 (xfr#11, to-chk=1/13)
migrations/000006_add_permissions.up.sql
 452 100% 441.41kB/s   0:00:00 (xfr#12, to-chk=0/13)
rsync -P ./remote/production/api.service greenlight@"45.55.49.87":~
api.service
 1,266 100%  0.00kB/s   0:00:00 (xfr#1, to-chk=0/1)
ssh -t greenlight@"45.55.49.87" '\
    migrate -path ~/migrations -database $GREENLIGHT_DB_DSN up \
    && sudo mv ~/api.service /etc/systemd/system/ \
    && sudo systemctl enable api \
    && sudo systemctl restart api \
'
no change
[sudo] password for greenlight:
Created symlink /etc/systemd/system/multi-user.target.wants/api.service → /etc/systemd/system/api.service.
Connection to 45.55.49.87 closed.

```

Next let's connect to the droplet and check the status of our new `api` service using the `sudo systemctl status api` command:

```
$ make production/connect
greenlight@greenlight-production:~$ sudo systemctl status api
● api.service - Greenlight API service
   Loaded: loaded (/etc/systemd/system/api.service; enabled; vendor preset: enabled)
   Active: active (running) since Mon 2021-04-19 03:24:35 EDT; 2min 50s ago
     Main PID: 6997 (api)
        Tasks: 6 (limit: 1136)
       Memory: 1.8M
      CGroup: /system.slice/api.service
              └─6997 /home/greenlight/api -port=4000 -db-dsn=postgres://greenlight:pa55word1234@localhost/greenlight -env=prod

Apr 19 03:24:35 greenlight-production systemd[1]: Starting Greenlight API service...
Apr 19 03:24:35 greenlight-production systemd[1]: Started Greenlight API service.
Apr 19 03:24:35 greenlight-production api[6997]: {"level":"INFO","time":"2021-04-19T07:24:35Z", ...}
Apr 19 03:24:35 greenlight-production api[6997]: {"level":"INFO","time":"2021-04-19T07:24:35Z", ...}
```

Great! This confirms that our **api** service is running successfully in the background and, in my case, that it has the PID (process ID) **6997**.

Out of interest, let's also quickly list the running processes for our **greenlight** user:

```
greenlight@greenlight-production:~$ ps -U greenlight
  PID TTY          TIME CMD
 6997 ?            00:00:00 api
 7043 ?            00:00:00 systemd
 7048 ?            00:00:00 (sd-pam)
 7117 ?            00:00:00 sshd
 7118 pts/0        00:00:00 bash
 7148 pts/0        00:00:00 ps
```

That tallies — we can see the **api** process listed here with the PID **6997**, confirming that it is indeed our **greenlight** user who is running the **api** binary.

Lastly, let's check from an external perspective and try making a request to our healthcheck endpoint again, either in a browser or with **curl**. You should get a successful response like so:

```
$ curl 45.55.49.87:4000/v1/healthcheck
{
  "status": "available",
  "system_info": {
    "environment": "production",
    "version": "v1.0.0-1-gf27fd0f"
  }
}
```

Restarting after reboot

By running the **systemctl enable api** command in our makefile after we copied across the

systemd unit file, our API service should be started up automatically after the droplet is rebooted.

If you like you can verify this yourself. While you're connected to the droplet via SSH, go ahead and reboot it (make sure you are connected to the droplet and don't accidentally reboot your local machine!).

```
greenlight@greenlight-production:~$ sudo reboot
greenlight@greenlight-production:~$ Connection to 45.55.49.87 closed by remote host.
Connection to 45.55.49.87 closed.
make: *** [Makefile:91: production/connect] Error 255
```

Wait a minute for the reboot to complete and for the droplet to come back online. Then you should be able to reconnect and use `systemctl status` to check that the service is running again. Like so:

```
$ make production/connect
greenlight@greenlight-production:~$ sudo systemctl status api.service
[sudo] password for greenlight:
● api.service - Greenlight API service
   Loaded: loaded (/etc/systemd/system/api.service; enabled; vendor preset: enabled)
   Active: active (running) since Mon 2021-04-19 03:36:26 EDT; 16s ago
     Main PID: 774 (api)
        Tasks: 5 (limit: 1136)
       Memory: 8.7M
      CGroup: /system.slice/api.service
              └─774 /home/greenlight/api -port=4000 -db-dsn=postgres://greenlight:pa55word1234@localhost/greenlight -env=produ

Apr 19 03:36:26 greenlight-production systemd[1]: Starting Greenlight API service...
Apr 19 03:36:26 greenlight-production systemd[1]: Started Greenlight API service.
Apr 19 03:36:26 greenlight-production api[774]: {"level":"INFO","time":"2021-04-19T07:36:26Z", ...}
Apr 19 03:36:26 greenlight-production api[774]: {"level":"INFO","time":"2021-04-19T07:36:26Z", ...}
```

Disable port 4000

If you've been following along with the steps in this section of the book, let's revert the temporary firewall change that we made earlier and disallow traffic on port 4000 again.

```
greenlight@greenlight-production:~$ sudo ufw delete allow 4000/tcp
Rule deleted
Rule deleted (v6)
greenlight@greenlight-production:~$ sudo ufw status
Status: active

To Action From
--
22 ALLOW Anywhere
80/tcp ALLOW Anywhere
443/tcp ALLOW Anywhere
22 (v6) ALLOW Anywhere (v6)
80/tcp (v6) ALLOW Anywhere (v6)
443/tcp (v6) ALLOW Anywhere (v6)
```

Additional Information

Listening on a restricted port

If you're *not* planning to run your application behind a reverse proxy, and want to listen for requests directly on port 80 or 443, you'll need to set up your unit file so that the service has the `CAP_NET_BIND_SERVICE` capability (which will allow it to bind to a restricted port). For example:

```
[Unit]
Description=Greenlight API service

After=postgresql.service
After=network-online.target
Wants=network-online.target

StartLimitIntervalSec=600
StartLimitBurst=5

[Service]
Type=exec
User=greenlight
Group=greenlight
CapabilityBoundingSet=CAP_NET_BIND_SERVICE
AmbientCapabilities=CAP_NET_BIND_SERVICE
EnvironmentFile=/etc/environment
WorkingDirectory=/home/greenlight
ExecStart=/home/greenlight/api -port=80 -db-dsn=${GREENLIGHT_DB_DSN} -env=production

Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target
```

Viewing logs

It's possible to view the logs for your background service using the `journalctl` command, like so:

```
$ sudo journalctl -u api
-- Logs begin at Sun 2021-04-18 14:55:45 EDT, end at Mon 2021-04-19 03:39:55 EDT. --
Apr 19 03:24:35 greenlight-production systemd[1]: Starting Greenlight API service...
Apr 19 03:24:35 greenlight-production systemd[1]: Started Greenlight API service.
Apr 19 03:24:35 greenlight-production api[6997]: {"level":"INFO","time":"2021-04-19T07:24:35Z", ...}
Apr 19 03:24:35 greenlight-production api[6997]: {"level":"INFO","time":"2021-04-19T07:24:35Z", ...}
```

The `journalctl` command is really powerful and offers a wide variety of parameters that you can use to filter your log messages and customize the formatting. [This article](#) provides a great introduction to the details of `journalctl` and is well worth a read.

Configuring the SMTP provider

Our unit file is currently set up to start our API with the following command:

```
ExecStart=/home/greenlight/api -port=4000 -db-dsn=${GREENLIGHT_DB_DSN} -env=production
```

It's important to remember that apart from the `port`, `db-dsn` and `env` flags that we're specifying here, our application will still be using the default values for the other settings which are hardcoded into the `cmd/api/main.go` file — including the SMTP credentials for your Mailtrap inbox. Under normal circumstances, you would want to set your production SMTP credentials as part of this command in the unit file too.

Additional unit file options

Systemd unit files offer a huge range of configuration options, and we've only just scratched the surface in this chapter. For more information, the [Understanding Systemd Units and Unit Files](#) article is a really good overview, and you can find comprehensive (albeit dense) information in the [man pages](#).

There is also [this gist](#) which talks through the various security options that you can use to help harden your service.

Using Caddy as a Reverse Proxy

We're now in the position where we have our Greenlight API application running as a background service on our droplet, and listening for HTTP requests on port **4000**. And we also have Caddy running as a background service and listening for HTTP requests on port **80**.

So the next step in setting up our production environment is to configure Caddy to act as a *reverse proxy* and forward any HTTP requests that it receives onward to our API.

The simplest way to configure Caddy is to create a *Caddyfile* — which contains a series of rules describing what we want Caddy to do. If you're following along, please go ahead and create a new **remote/production/Caddyfile** file in your project directory:

```
$ touch remote/production/Caddyfile
```

And then add the following content, making sure to replace the IP address with the address of your own droplet:

```
File: remote/production/Caddyfile
```

```
http://45.55.49.87 {  
  reverse_proxy localhost:4000  
}
```

As you can probably guess by looking at it, this rule tells Caddy that we want it to listen for HTTP requests to **45.55.49.87** and then act a reverse proxy, forwarding the request to port **4000** on the local machine (where our API is listening).

Hint: Caddyfiles have *a lot* of available settings and we'll demonstrate some of the more common ones as we progress through this chapter. But if you plan on using Caddy in your own production projects, then I recommend reading through the [official Caddyfile documentation](#), which is excellent.

The next thing we need to do is upload this Caddyfile to our droplet and reload Caddy for the changes to take effect.

To manage this, let's update our `production/deploy/api` rule. This will follow the same basic pattern that we used in the previous chapter for uploading our `api.service` file, but instead we'll copy our Caddyfile to `/etc/caddy/Caddyfile` on the server.

Like so:

File: Makefile

```
# ===== #
# PRODUCTION
# ===== #

production_host_ip = '45.55.49.87'

...

## production/deploy/api: deploy the api to production
.PHONY: production/deploy/api
production/deploy/api:
  rsync -P ./bin/linux_amd64/api greenlight@${production_host_ip}:~
  rsync -rP --delete ./migrations greenlight@${production_host_ip}:~
  rsync -P ./remote/production/api.service greenlight@${production_host_ip}:~
  rsync -P ./remote/production/Caddyfile greenlight@${production_host_ip}:~
  ssh -t greenlight@${production_host_ip} '\
    migrate -path ~/migrations -database $$GREENLIGHT_DB_DSN up \
    && sudo mv ~/api.service /etc/systemd/system/ \
    && sudo systemctl enable api \
    && sudo systemctl restart api \
    && sudo mv ~/Caddyfile /etc/caddy/ \
    && sudo systemctl reload caddy \
  '
```

Once you've made that change, go ahead and run the rule to deploy the Caddyfile into production:

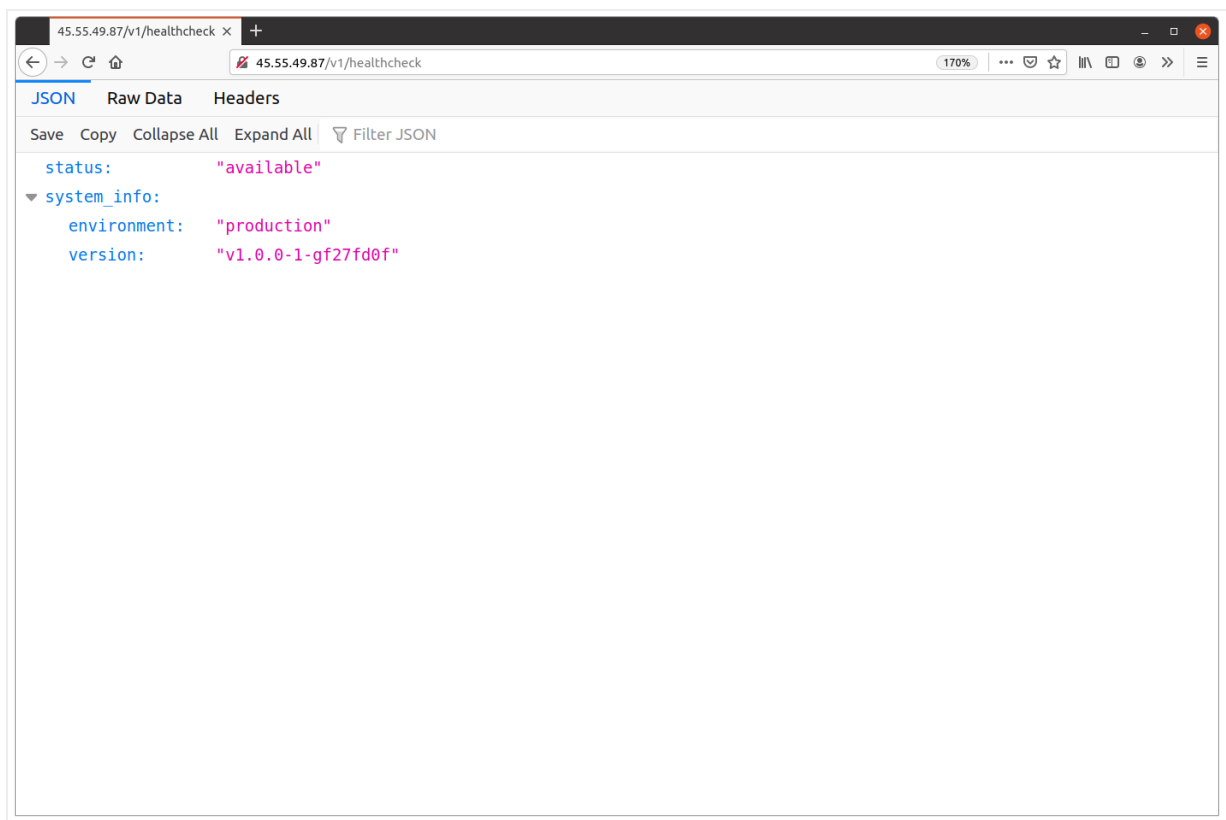
```

$ make production/deploy/api
rsync -P ./bin/linux_amd64/api greenlight@"45.55.49.87":~
api
  7,634,944 100%  70.67MB/s   0:00:00 (xfr#1, to-chk=0/1)
rsync -rP --delete ./migrations greenlight@"45.55.49.87":~
sending incremental file list
migrations/000001_create_movies_table.down.sql
  28 100%  0.00kB/s   0:00:00 (xfr#1, to-chk=11/13)
migrations/000001_create_movies_table.up.sql
  286 100% 279.30kB/s   0:00:00 (xfr#2, to-chk=10/13)
migrations/000002_add_movies_check_constraints.down.sql
  198 100% 193.36kB/s   0:00:00 (xfr#3, to-chk=9/13)
migrations/000002_add_movies_check_constraints.up.sql
  289 100% 282.23kB/s   0:00:00 (xfr#4, to-chk=8/13)
migrations/000003_add_movies_indexes.down.sql
  78 100%  76.17kB/s   0:00:00 (xfr#5, to-chk=7/13)
migrations/000003_add_movies_indexes.up.sql
  170 100% 166.02kB/s   0:00:00 (xfr#6, to-chk=6/13)
migrations/000004_create_users_table.down.sql
  27 100%  26.37kB/s   0:00:00 (xfr#7, to-chk=5/13)
migrations/000004_create_users_table.up.sql
  294 100% 287.11kB/s   0:00:00 (xfr#8, to-chk=4/13)
migrations/000005_create_tokens_table.down.sql
  28 100%  27.34kB/s   0:00:00 (xfr#9, to-chk=3/13)
migrations/000005_create_tokens_table.up.sql
  203 100% 198.24kB/s   0:00:00 (xfr#10, to-chk=2/13)
migrations/000006_add_permissions.down.sql
  73 100%  71.29kB/s   0:00:00 (xfr#11, to-chk=1/13)
migrations/000006_add_permissions.up.sql
  452 100% 441.41kB/s   0:00:00 (xfr#12, to-chk=0/13)
rsync -P ./remote/production/api.service greenlight@"45.55.49.87":~
api.service
  1,266 100%  0.00kB/s   0:00:00 (xfr#1, to-chk=0/1)
rsync -P ./remote/production/Caddyfile greenlight@"45.55.49.87":~
Caddyfile
  293 100%  0.00kB/s   0:00:00 (xfr#1, to-chk=0/1)
ssh -t greenlight@"45.55.49.87" '\
  migrate -path ~/migrations -database $GREENLIGHT_DB_DSN up \
  && sudo mv ~/api.service /etc/systemd/system/ \
  && sudo systemctl enable api \
  && sudo systemctl restart api \
  && sudo mv ~/Caddyfile /etc/caddy/ \
  && sudo systemctl reload caddy \
'
no change
[sudo] password for greenlight:
Connection to 45.55.49.87 closed.

```

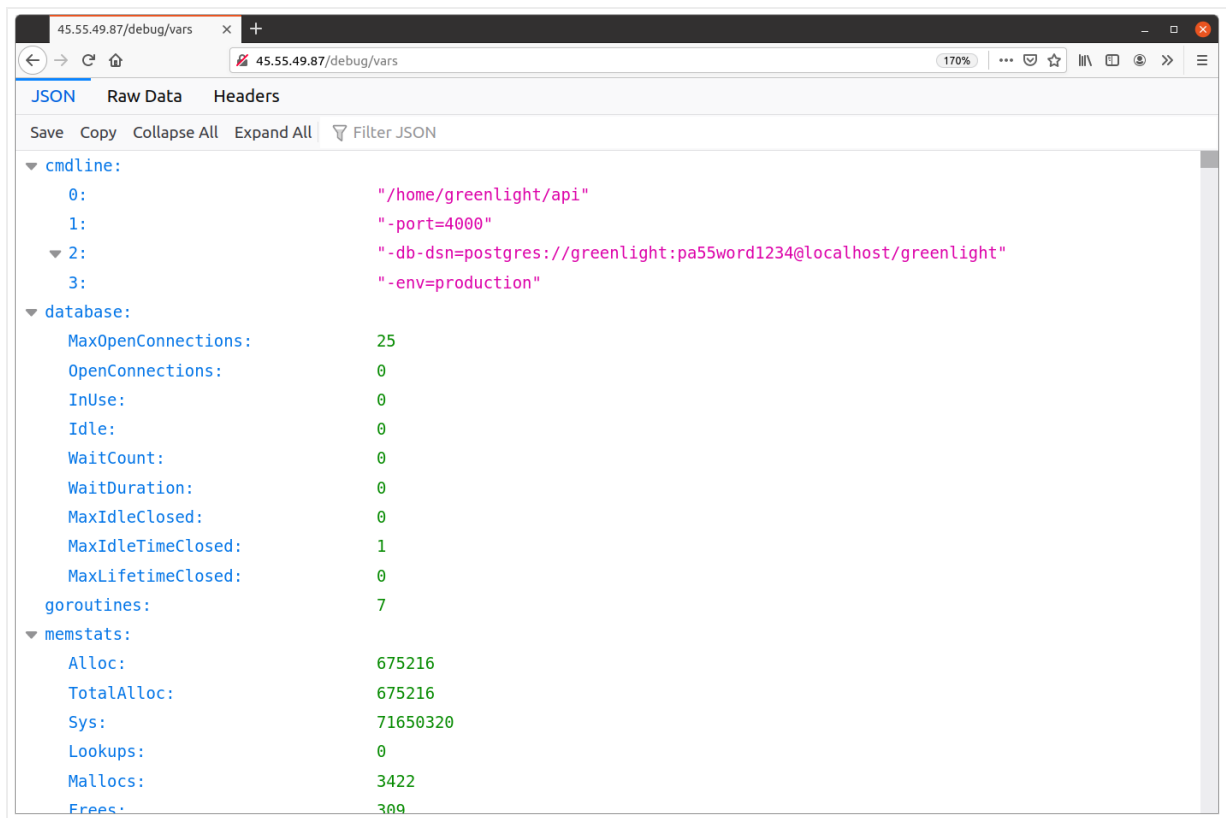
You should see that the Caddyfile is copied across and the reload executes cleanly without any errors.

At this point you can visit http://<your_droplet_ip>/v1/healthcheck in a web browser, and you should find that the request is successfully forwarded on from Caddy to our API. Like so:



Blocking access to application metrics

While we're in the browser, let's navigate to the `GET /debug/vars` endpoint which displays our application metrics. You should see a response similar to this:



As we mentioned earlier, it's really not a good idea for this sensitive information to be publicly accessible.

Fortunately, it's very easy to block access to this by adding a new `respond` directive to our Caddyfile like so:

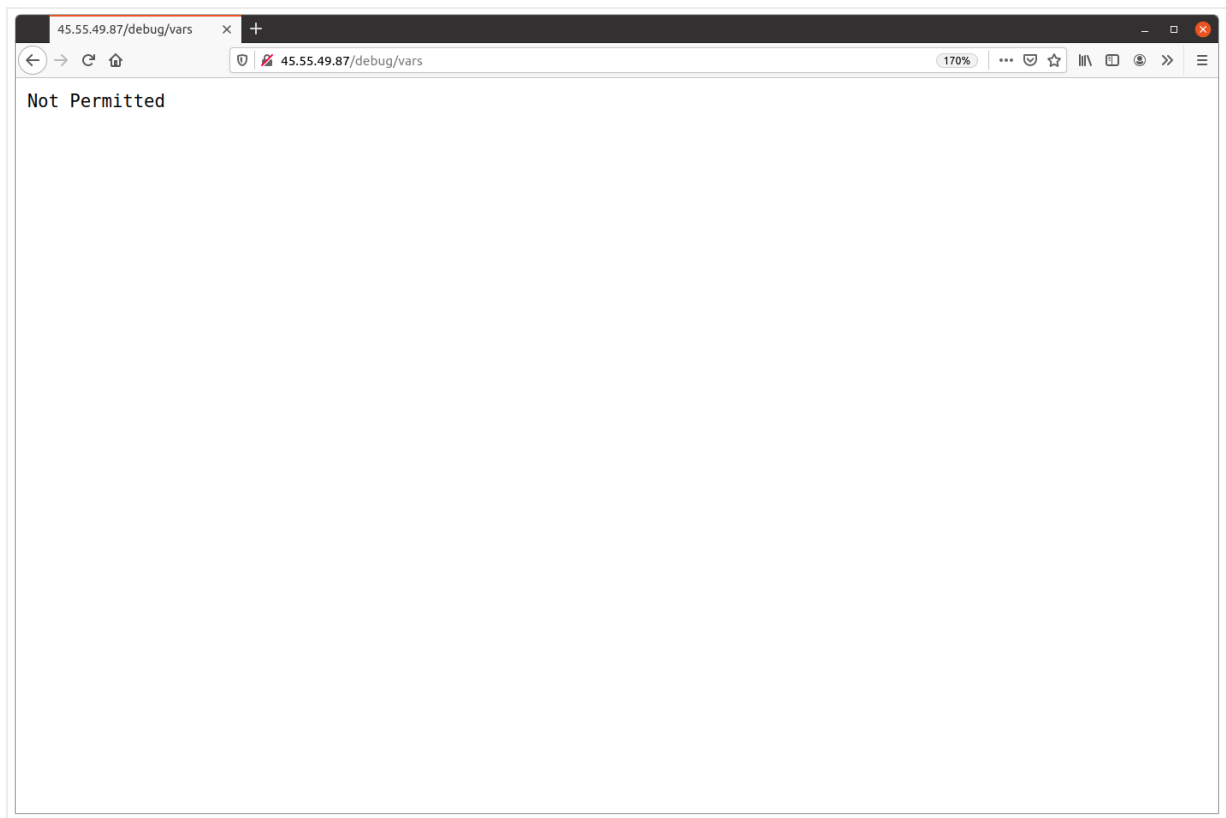
File: remote/production/Caddyfile

```
http://45.55.49.87 {  
  respond /debug/* "Not Permitted" 403  
  reverse_proxy localhost:4000  
}
```

With this new directive we're instructing Caddy to send a `403 Forbidden` response for all requests which have a URL path beginning `/debug/`.

Go ahead and deploy this change to production again, and when you refresh the page in your web browser you should find that it is now blocked.

```
$ make production/deploy/api
```



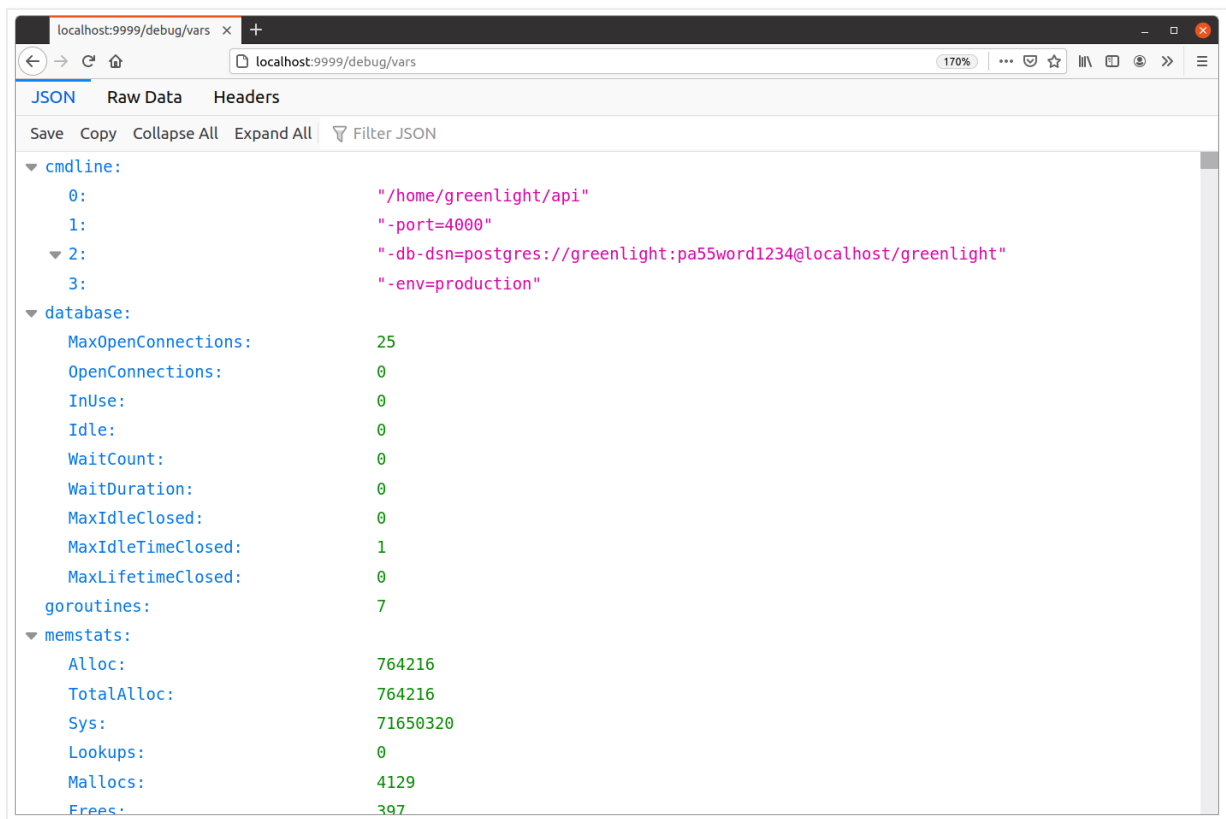
Although the metrics are no longer publicly accessible, you can still access them by connecting to your droplet via SSH and making a request to <http://localhost:4000/debug/vars>.

```
$ make production/connect
greenlight@greenlight-production:~$ curl http://localhost:4000/debug/vars
{
  "cmdline": "...",
  "database": "...",
  "goroutines": 7,
  "memstats": "...",
  "timestamp": 1618820037,
  "total_processing_time_us": 1305,
  "total_requests_received": 8,
  "total_responses_sent": 7,
  "total_responses_sent_by_status": {"200": 3, "404": 4},
  "version": "v1.0.0-1-gf27fd0f"
}
```

Or alternatively, you can open a SSH tunnel to the droplet and view them using a web browser on your *local machine*. For example, you could open an SSH tunnel between port **4000** on the droplet and port **9999** on your local machine by running the following command (make sure to replace *both* IP addresses with your own droplet IP):

```
$ ssh -L :9999:45.55.49.87:4000 greenlight@45.55.49.87
```

While that tunnel is active, you should be able to visit <http://localhost:9999/debug/vars> in your web browser and see your application metrics, like so:



Using a domain name

For the next step of our deployment we're going to configure Caddy so that we can access our droplet via a domain name, instead of needing to use the IP address.

If you want to follow along with this step you'll need access to a domain name and the ability to update the DNS records for that domain name. If you don't have a domain name already available that you can use, then you can get a free one via the [Freenom](#) service.

I'm going to use the domain `greenlight.alexedwards.net` in the sample code here, but you should swap this out for your own domain if you're following along.

The first thing you'll need to do is configure the DNS records for your domain name so that they contain an **A** record pointing to the IP address for your droplet. So in my case the DNS record would look like this:

A	greenlight.alexedwards.net	45.55.49.87

Note: If you're not sure how to alter your DNS records, your domain name registrar should provide guidance and documentation.

Once you've got the DNS record in place, the next task is to update the Caddyfile to use your domain name instead of your droplet's IP address. Go ahead and swap this out like so (remember to replace `greenlight.alexedwards.net` with your own domain name):

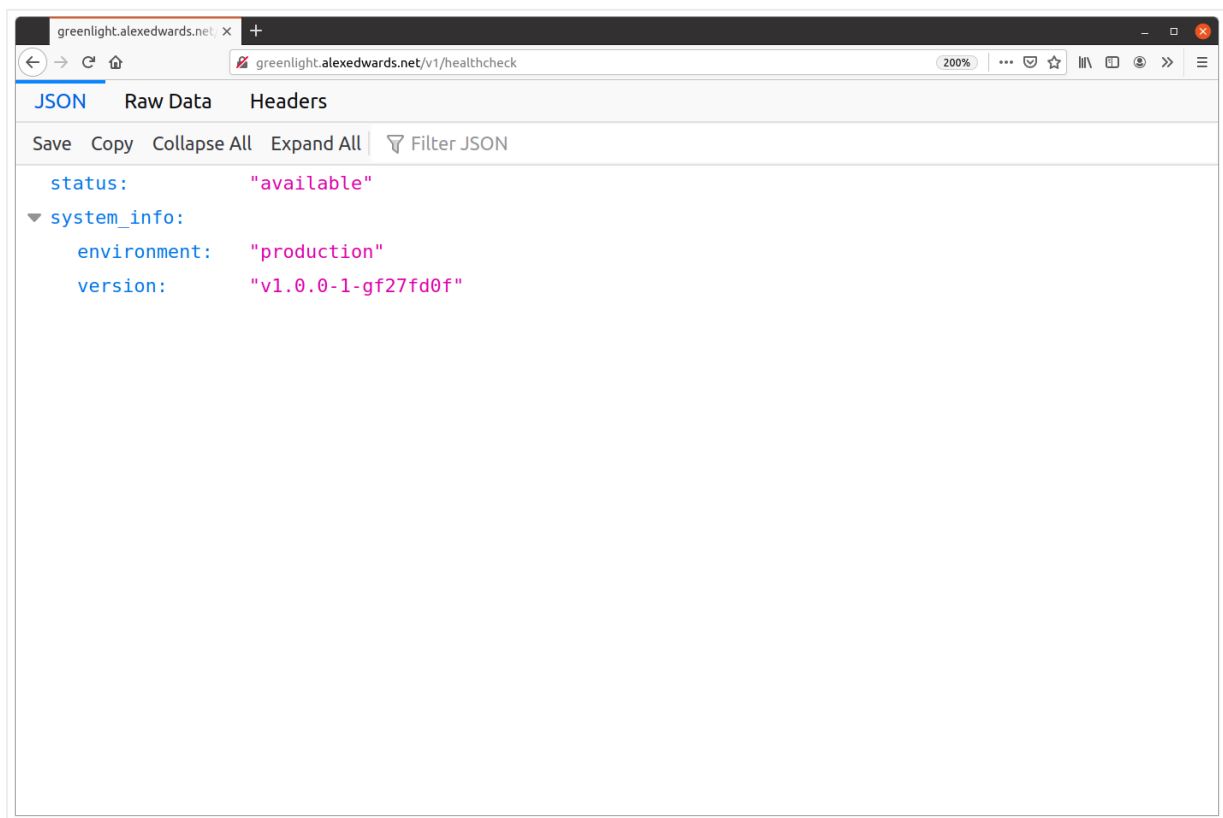
```
File: remote/production/Caddyfile

http://greenlight.alexedwards.net {
  respond /debug/* "Not Permitted" 403
  reverse_proxy localhost:4000
}
```

And then redeploy the Caddyfile to your droplet again:

```
$ make production/deploy/api
```

Once you've done that, you should now be able to access the API via your domain name by visiting `http://<your_domain_name>/v1/healthcheck` in your browser:



Enabling HTTPS

Now that we have a domain name set up we can utilize one of Caddy's headline features: *automatic HTTPS*.

Caddy will automatically handle provisioning and renewing TLS certificates for your domain via Let's Encrypt, as well as redirecting all HTTP requests to HTTPS. It's simple to set up, very robust, and saves you the overhead of needing to keep track of certificate renewals manually.

To enable this, we just need to update our **Caddyfile** so that it looks like this:

File: remote/production/Caddyfile

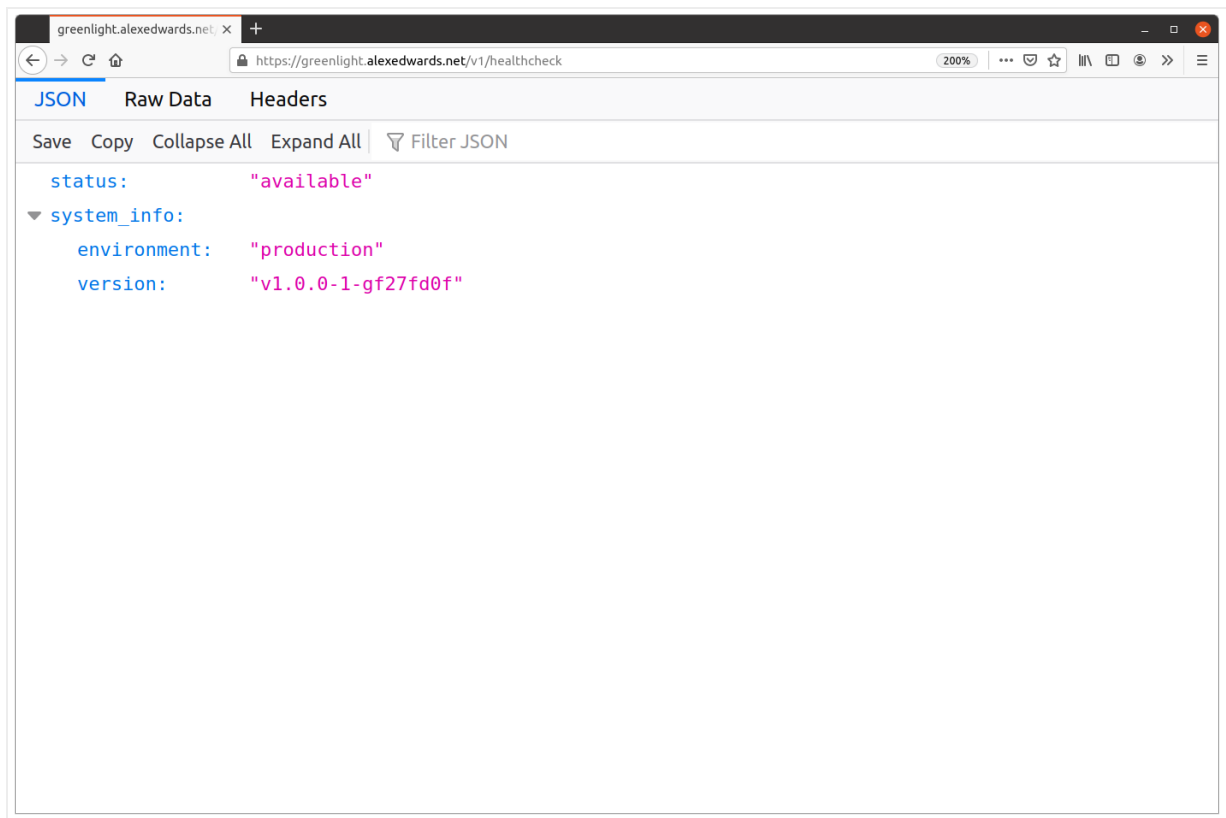
```
# Set the email address that should be used to contact you if there is a problem with
# your TLS certificates.
{
    email you@example.com
}

# Remove the http:// prefix from your site address.
greenlight.alexedwards.net {
    respond /debug/* "Not Permitted" 403
    reverse_proxy localhost:4000
}
```

For the final time, deploy this Caddyfile update to your droplet...

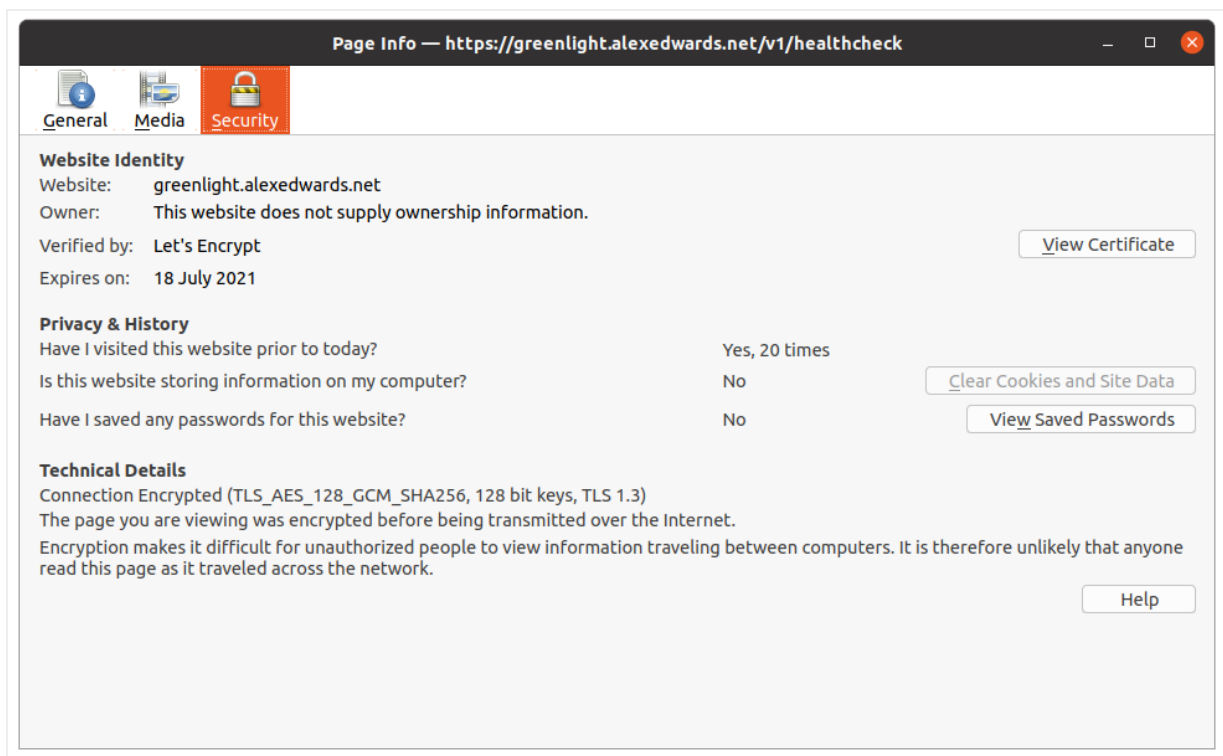
```
$ make production/deploy/api
```

And then when you refresh the page in your web browser, you should find that it is automatically redirected to a HTTPS version of the page.



Important: If you get an error when you refresh the page, wait a few seconds and then try again. Sometimes it can take Caddy a moment to complete the certificate set up for the very first time.

If you're using Firefox, you can also look at the page information in your browser by pressing **Ctrl+I**. It should look similar to this:



We can see from this that the connection has been successfully encrypted using TLS 1.3 and using the `TLS_AES_128_GCM_SHA256` cipher suite.

Lastly, if you want, you can use `curl` to try making a HTTP request to the application. You should see that this issues a `308 Permanent Redirect` to the HTTPS version of the application, like so:

```
$ curl -i http://greenlight.alexedwards.net
HTTP/1.1 308 Permanent Redirect
Connection: close
Location: https://greenlight.alexedwards.net/
Server: Caddy
Date: Mon, 19 Apr 2021 08:36:20 GMT
Content-Length: 0
```

Additional Information

Scaling the infrastructure

Before launching a new service, it's often useful to do a thought experiment and ask yourself: *What happens as traffic to the service increases? How would I manage it?*

For this project, there's a clear path that we could follow to scale up and adapt the

infrastructure to accommodate growth. Very roughly, the path looks something like this:

- Use single low-powered droplet running Caddy, PostgreSQL, and the Go application. This is what we currently have.
- ↓ Upgrade the droplet to have more CPU and/or memory as necessary.
- ↓ Move PostgreSQL to a separate droplet, or use a [managed database](#).
- ↓ Upgrade droplets/managed databases to have more CPU and/or memory as necessary.
- If the droplet running the Go application is a bottleneck:
 - ↓ Profile and optimize your Go application code.
 - ↓ Run the Go application on multiple droplets, behind a load balancer.
- If the PostgreSQL database is a bottleneck:
 - ↓ Profile and optimize the database settings and database queries.
 - ↓ If appropriate, cache the results of expensive/frequent database queries.
 - ↓ If appropriate, move some operations to an in-memory database such as Redis.
 - ↓ Start using read-only database replicas for queries where possible.
 - ↓ Start sharding the database.

In other projects, these steps and the order of them will probably be different. And in some cases, it may be sensible to jump straight into a more complex architecture straight away, based on business requirements, traffic estimates and load testing results.

But taking a short moment to do a thought experiment like this is good. If you can see a clear path, then it provides some positive reassurance that growth can be accommodated without you optimizing the infrastructure or application prematurely. Whereas if you *can't* see a clear path to accommodate growth, then it highlights a potential problem and it may be sensible to revisit your infrastructure choices.

Appendices

- [Managing Password Resets](#)
- [Creating Additional Activation Tokens](#)
- [Authentication with JSON Web Tokens](#)
- [JSON Encoding Nuances](#)
- [JSON Decoding Nuances](#)
- [Request Content Timeouts](#)

Managing Password Resets

If the API you're building is for a public audience, it's likely that you'll want to include functionality for a user to reset their password when they forget it.

To support this in your API, you could add the following two endpoints:

Method	URL Pattern	Handler	Action
POST	/v1/tokens/password-reset	createPasswordResetTokenHandler	Generate a new password reset token
PUT	/v1/users/password	updateUserPasswordHandler	Update the password for a specific user

And implement a workflow like this:

1. A client sends a request to the **POST /v1/tokens/password-reset** endpoint containing the email address of the user whose password they want to reset.
2. If a user with that email address exists in the **users** table, *and the user has already confirmed their email address by activating*, then generate a cryptographically-secure high-entropy random token.
3. Store a hash of this token in the **tokens** table, alongside the user ID and a short (30-60 minute) expiry time for the token.
4. Send the original (unhashed) token to the user in an email.
5. If the owner of the email address didn't request a password reset token, they can ignore the email.
6. Otherwise, they can submit the token to the **PUT /v1/users/password** endpoint along with their new password. If the hash of the token exists in the **tokens** table and hasn't expired, then generate a bcrypt hash of the new password and update the user's record.
7. Delete all existing password reset tokens for the user.

In our codebase, you could implement this workflow by creating a new **password-reset** token scope:

File: internal/data/tokens.go

```
package data

...

const (
    ScopeActivation    = "activation"
    ScopeAuthentication = "authentication"
    ScopePasswordReset = "password-reset"
)

...
```

And then adding the following two handlers:

File: cmd/api/tokens.go

```
package main

...

// Generate a password reset token and send it to the user's email address.
func (app *application) createPasswordResetTokenHandler(w http.ResponseWriter, r *http.Request) {
    // Parse and validate the user's email address.
    var input struct {
        Email string `json:"email"`
    }

    err := app.readJSON(w, r, &input)
    if err != nil {
        app.badRequestResponse(w, r, err)
        return
    }

    v := validator.New()

    if data.ValidateEmail(v, input.Email); !v.Valid() {
        app.failedValidationResponse(w, r, v.Errors)
        return
    }

    // Try to retrieve the corresponding user record for the email address. If it can't
    // be found, return an error message to the client.
    user, err := app.models.Users.GetByEmail(input.Email)
    if err != nil {
        switch {
        case errors.Is(err, data.ErrRecordNotFound):
            v.AddError("email", "no matching email address found")
            app.failedValidationResponse(w, r, v.Errors)
        default:
            app.serverErrorResponse(w, r, err)
        }
        return
    }

    // Return an error message if the user is not activated.
    if !user.Activated {
        v.AddError("email", "user account must be activated")
        app.failedValidationResponse(w, r, v.Errors)
        return
    }
}
```

```

}

// Otherwise, create a new password reset token with a 45-minute expiry time.
token, err := app.models.Tokens.New(user.ID, 45*time.Minute, data.ScopePasswordReset)
if err != nil {
    app.serverErrorResponse(w, r, err)
    return
}

// Email the user with their password reset token.
app.background(func() {
    data := map[string]interface{}{
        "passwordResetToken": token.Plaintext,
    }

    // Since email addresses MAY be case sensitive, notice that we are sending this
    // email using the address stored in our database for the user --- not to the
    // input.Email address provided by the client in this request.
    err = app.mailer.Send(user.Email, "token_password_reset.tmpl", data)
    if err != nil {
        app.logger.PrintError(err, nil)
    }
})

// Send a 202 Accepted response and confirmation message to the client.
env := envelope{"message": "an email will be sent to you containing password reset instructions"}

err = app.writeJSON(w, http.StatusAccepted, env, nil)
if err != nil {
    app.serverErrorResponse(w, r, err)
}
}
}

```

File: cmd/api/users.go

```

package main

...

// Verify the password reset token and set a new password for the user.
func (app *application) updateUserPasswordHandler(w http.ResponseWriter, r *http.Request) {
    // Parse and validate the user's new password and password reset token.
    var input struct {
        Password      string `json:"password"`
        TokenPlaintext string `json:"token"`
    }

    err := app.readJSON(w, r, &input)
    if err != nil {
        app.badRequestResponse(w, r, err)
        return
    }

    v := validator.New()

    data.ValidatePasswordPlaintext(v, input.Password)
    data.ValidateTokenPlaintext(v, input.TokenPlaintext)

    if !v.Valid() {
        app.failedValidationResponse(w, r, v.Errors)
        return
    }

    // Retrieve the details of the user associated with the password reset token,
    // then use the details to update the user's password.
}

```



```

// returning an error message if no matching record was found.
user, err := app.models.Users.GetForToken(data.ScopePasswordReset, input.TokenPlaintext)
if err != nil {
    switch {
    case errors.Is(err, data.ErrRecordNotFound):
        v.AddError("token", "invalid or expired password reset token")
        app.failedValidationResponse(w, r, v.Errors)
    default:
        app.serverErrorResponse(w, r, err)
    }
    return
}

// Set the new password for the user.
err = user.Password.Set(input.Password)
if err != nil {
    app.serverErrorResponse(w, r, err)
    return
}

// Save the updated user record in our database, checking for any edit conflicts as
// normal.
err = app.models.Users.Update(user)
if err != nil {
    switch {
    case errors.Is(err, data.ErrEditConflict):
        app.editConflictResponse(w, r)
    default:
        app.serverErrorResponse(w, r, err)
    }
    return
}

// If everything was successful, then delete all password reset tokens for the user.
err = app.models.Tokens.DeleteAllForUser(data.ScopePasswordReset, user.ID)
if err != nil {
    app.serverErrorResponse(w, r, err)
    return
}

// Send the user a confirmation message.
env := envelope{"message": "your password was successfully reset"}

err = app.writeJSON(w, http.StatusOK, env, nil)
if err != nil {
    app.serverErrorResponse(w, r, err)
}
}

```

You should also create an `internal/mailer/templates/token_password_reset.tmpl` file containing the email templates for the password reset email, similar to this:

File: internal/mailer/templates/token_password_reset.tpl

```
{{define "subject"}}Reset your Greenlight password{{end}}
```

```
{{define "plainBody"}}
```

Hi,

Please send a `PUT /v1/users/password` request with the following JSON body to set a new password:

```
{"password": "your new password", "token": "{{.passwordResetToken}}"}{
```

Please note that this is a one-time use token and it will expire in 45 minutes. If you need another token please make a `POST /v1/tokens/password-reset` request.

Thanks,

The Greenlight Team

```
{{end}}
```

```
{{define "htmlBody"}}
```

```
<!doctype html>
```

```
<html>
```

```
<head>
```

```
<meta name="viewport" content="width=device-width" />
```

```
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
```

```
</head>
```

```
<body>
```

```
<p>Hi,</p>
```

```
<p>Please send a <code>PUT /v1/users/password</code> request with the following JSON body to set a new password:</p>
```

```
<pre><code>
```

```
{"password": "your new password", "token": "{{.passwordResetToken}}"}{
```

```
</code></pre>
```

```
<p>Please note that this is a one-time use token and it will expire in 45 minutes.
```

```
If you need another token please make a <code>POST /v1/tokens/password-reset</code> request.</p>
```

```
<p>Thanks,</p>
```

```
<p>The Greenlight Team</p>
```

```
</body>
```

```
</html>
```

```
{{end}}
```

And add the necessary routes to the `cmd/api/routes.go` file:

File: cmd/api/routes.go

```
package main

...

func (app *application) routes() http.Handler {
    router := httprouter.New()

    router.NotFound = http.HandlerFunc(app.notFoundResponse)
    router.MethodNotAllowed = http.HandlerFunc(app.methodNotAllowedResponse)

    router.HandleFunc(http.MethodGet, "/v1/healthcheck", app.healthcheckHandler)

    router.HandleFunc(http.MethodGet, "/v1/movies", app.requirePermission("movies:read", app.listMoviesHandler))
    router.HandleFunc(http.MethodPost, "/v1/movies", app.requirePermission("movies:write", app.createMovieHandler))
    router.HandleFunc(http.MethodGet, "/v1/movies/:id", app.requirePermission("movies:read", app.showMovieHandler))
    router.HandleFunc(http.MethodPatch, "/v1/movies/:id", app.requirePermission("movies:write", app.updateMovieHandler))
    router.HandleFunc(http.MethodDelete, "/v1/movies/:id", app.requirePermission("movies:write", app.deleteMovieHandler))

    router.HandleFunc(http.MethodPost, "/v1/users", app.registerUserHandler)
    router.HandleFunc(http.MethodPut, "/v1/users/activated", app.activateUserHandler)
    // Add the PUT /v1/users/password endpoint.
    router.HandleFunc(http.MethodPut, "/v1/users/password", app.updateUserPasswordHandler)

    router.HandleFunc(http.MethodPost, "/v1/tokens/authentication", app.createAuthenticationTokenHandler)
    // Add the POST /v1/tokens/password-reset endpoint.
    router.HandleFunc(http.MethodPost, "/v1/tokens/password-reset", app.createPasswordResetTokenHandler)

    router.Handler(http.MethodGet, "/debug/vars", expvar.Handler())

    return app.metrics(app.recoverPanic(app.enableCORS(app.rateLimit(app.authenticate(router)))))
}
```

Once those things are all in place, you should be able to request a new password reset token by making a request like this:

```
$ curl -X POST -d '{"email": "alice@example.com"}' localhost:4000/v1/tokens/password-reset
{
  "message": "an email will be sent to you containing password reset instructions"
}
```

And then you can initiate the actual password change by sending a request containing the token received in the email. For example:

```
$ BODY='{"password": "your new password", "token": "Y7QCRZ7FW0WYXLAC2VYOLIPY"}'
$ curl -X PUT -d "$BODY" localhost:4000/v1/users/password
{
  "message": "your password was successfully reset"
}
```

Additional Information

Web application workflow

Just like the activation workflow we looked at earlier, if your API is the backend to a website (rather than a completely standalone service) you can tweak this password reset workflow to make it more intuitive for users.

For example, you could ask the user to enter the token into a form on your website along with their new password, and use some JavaScript to submit the form contents to your `PUT /v1/users/password` endpoint. The email to support that workflow could look something like this:

```
Hi,  
  
To reset your password please visit https://example.com/users/password and enter the  
following secret code along with your new password:  
  
-----  
Y7QCRZ7FW0WYLXLAOC2VYOLIPY  
-----  
  
Please note that this code will expire in 45 minutes. If you need another code please  
visit https://example.com/tokens/password-reset.  
  
Thanks,  
  
The Greenlight Team
```

Alternatively, if you don't want the user to copy-and-paste a token, you could ask them to click a link containing the token which takes them to a page on your website. Similar to this:

```
Hi,  
  
To reset your password please click the following link:  
  
https://example.com/users/password?token=Y7QCRZ7FW0WYLXLAOC2VYOLIPY  
  
Please note that this link will expire in 45 minutes. If you need another password reset  
link please visit https://example.com/tokens/password-reset.  
  
Thanks,  
  
The Greenlight Team
```

This page can then display a form in which the user enters their new password, and some JavaScript on the webpage should then extract the token from the URL and submit it to the `PUT /v1/users/password` endpoint along with the new password. Again, if you go with this second option, you need to take steps to avoid the token being leaked in a referrer header.

Creating Additional Activation Tokens

You may also want to add a standalone endpoint to your API for generating activation tokens. This can be useful in scenarios where a user doesn't activate their account in time, or they never receive their welcome email.

In this appendix we'll quickly run through the code for doing that, and add the following endpoint:

Method	URL Pattern	Handler	Action
POST	/v1/tokens/activation	createActivationTokenHandler	Generate a new activation token

The code for the `createActivationTokenHandler` handler should look like this:

File: cmd/api/tokens.go

...

```
func (app *application) createActivationTokenHandler(w http.ResponseWriter, r *http.Request) {
    // Parse and validate the user's email address.
    var input struct {
        Email string `json:"email"`
    }

    err := app.readJSON(w, r, &input)
    if err != nil {
        app.badRequestResponse(w, r, err)
        return
    }

    v := validator.New()

    if data.ValidateEmail(v, input.Email); !v.Valid() {
        app.failedValidationResponse(w, r, v.Errors)
        return
    }

    // Try to retrieve the corresponding user record for the email address. If it can't
    // be found, return an error message to the client.
    user, err := app.models.Users.GetByEmail(input.Email)
    if err != nil {
        switch {
        case errors.Is(err, data.ErrRecordNotFound):
            v.AddError("email", "no matching email address found")
            app.failedValidationResponse(w, r, v.Errors)
        default:
            app.serverErrorResponse(w, r, err)
        }
        return
    }
}
```

```

// Return an error if the user has already been activated.
if user.Activated {
    v.AddError("email", "user has already been activated")
    app.failedValidationResponse(w, r, v.Errors)
    return
}

// Otherwise, create a new activation token.
token, err := app.models.Tokens.New(user.ID, 3*24*time.Hour, data.ScopeActivation)
if err != nil {
    app.serverErrorResponse(w, r, err)
    return
}

// Email the user with their additional activation token.
app.background(func() {
    data := map[string]interface{}{
        "activationToken": token.Plaintext,
    }

    // Since email addresses MAY be case sensitive, notice that we are sending this
    // email using the address stored in our database for the user --- not to the
    // input.Email address provided by the client in this request.
    err = app.mailer.Send(user.Email, "token_activation.tmpl", data)
    if err != nil {
        app.logger.PrintError(err, nil)
    }
})

// Send a 202 Accepted response and confirmation message to the client.
env := envelope{"message": "an email will be sent to you containing activation instructions"}

err = app.writeJSON(w, http.StatusAccepted, env, nil)
if err != nil {
    app.serverErrorResponse(w, r, err)
}
}

```

You'll also need to create an `internal/mailer/templates/token_activation.tmpl` file containing the necessary email templates, similar to this:

File: internal/mailer/templates/token_activation.tmpl

```
{{define "subject"}}Activate your Greenlight account{{end}}

{{define "plainBody"}}
Hi,

Please send a `PUT /v1/users/activated` request with the following JSON body to activate your account:

{"token": "{{.activationToken}}"}

Please note that this is a one-time use token and it will expire in 3 days.

Thanks,

The Greenlight Team
{{end}}

{{define "htmlBody"}}
<!doctype html>
<html>
  <head>
    <meta name="viewport" content="width=device-width" />
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  </head>
  <body>
    <p>Hi,</p>
    <p>Please send a <code>PUT /v1/users/activated</code> request with the following JSON body to activate your account:</p>
    <pre><code>
{"token": "{{.activationToken}}"}
</code></pre>
    <p>Please note that this is a one-time use token and it will expire in 3 days.</p>
    <p>Thanks,</p>
    <p>The Greenlight Team</p>
  </body>
</html>
{{end}}
```

And update the `cmd/api/routes.go` file to include the new endpoint:

File: cmd/api/routes.go

```
package main

...

func (app *application) routes() http.Handler {
    router := httprouter.New()

    router.NotFound = http.HandlerFunc(app.notFoundResponse)
    router.MethodNotAllowed = http.HandlerFunc(app.methodNotAllowedResponse)

    router.HandleFunc(http.MethodGet, "/v1/healthcheck", app.healthcheckHandler)

    router.HandleFunc(http.MethodGet, "/v1/movies", app.requirePermission("movies:read", app.listMoviesHandler))
    router.HandleFunc(http.MethodPost, "/v1/movies", app.requirePermission("movies:write", app.createMovieHandler))
    router.HandleFunc(http.MethodGet, "/v1/movies/:id", app.requirePermission("movies:read", app.showMovieHandler))
    router.HandleFunc(http.MethodPatch, "/v1/movies/:id", app.requirePermission("movies:write", app.updateMovieHandler))
    router.HandleFunc(http.MethodDelete, "/v1/movies/:id", app.requirePermission("movies:write", app.deleteMovieHandler))

    router.HandleFunc(http.MethodPost, "/v1/users", app.registerUserHandler)
    router.HandleFunc(http.MethodPut, "/v1/users/activated", app.activateUserHandler)
    router.HandleFunc(http.MethodPut, "/v1/users/password", app.updateUserPasswordHandler)

    router.HandleFunc(http.MethodPost, "/v1/tokens/authentication", app.createAuthenticationTokenHandler)
    // Add the POST /v1/tokens/activation endpoint.
    router.HandleFunc(http.MethodPost, "/v1/tokens/activation", app.createActivationTokenHandler)
    router.HandleFunc(http.MethodPost, "/v1/tokens/password-reset", app.createPasswordResetTokenHandler)

    router.Handler(http.MethodGet, "/debug/vars", expvar.Handler())

    return app.metrics(app.recoverPanic(app.enableCORS(app.rateLimit(app.authenticate(router)))))
}
```

Now those things are all in place, a user can request a new activation token by submitting their email address like this:

```
$ curl -X POST -d '{"email": "bob@example.com"}' localhost:4000/v1/tokens/activation
{
  "message": "an email will be sent to you containing activation instructions"
}
```

The token will be sent to them in an email, and they can then submit the token to the `PUT /v1/users/activated` endpoint to activate, in exactly the same way as if they received the token in their welcome email.

If you implement an endpoint like this, it's important to note that this would allow users to potentially have multiple valid activation tokens 'on the go' at any one time. That's fine — but you just need to make sure that you delete *all* the activation tokens for a user once they've successfully activated (not just the token that they used).

And again, if your API is the backend for a website, then you can tweak the emails and workflow to make it more intuitive by using the same kind of patterns that we've talked

about previously.

Authentication with JSON Web Tokens

In this appendix we're going to switch the authentication process for our API to use JSON Web Tokens (JWTs).

Important: Using JWTs for this particular application doesn't have any benefits over our current 'stateful token' approach. The JWT approach is more complex, ultimately requires the same number of database lookups, and we lose the ability to revoke tokens. For those reasons, it doesn't make much sense to use them here. But I still want to explain the pattern for two reasons:

- JWTs have a lot of mind-share, and as a developer it's likely that you will run into existing codebases that use them for authentication, or that outside influences mean you are forced to use them.
- It's worth understanding how they work in case you need to implement APIs that require *delegated authentication*, which *is* a scenario that they're useful in.

Note: If you're completely new to JWTs, I recommend reading the [Introduction to JSON Web Tokens](#) article as a starting point.

As we briefly explained earlier in the book, JWTs are a type of *stateless token*. They contain a set of *claims* which are signed (using either a symmetric or asymmetric signing algorithm) and then encoded using base-64. In our case, we'll use JWTs to carry a *subject* claim containing the ID of an authenticated user.

There are a few different packages available which make working with JWTs relatively simple in Go. In most cases the [pascaldekloe/jwt](#) package is a good choice — it has a clear and simple API, and is designed to avoid a couple of the major JWT security vulnerabilities by default.

If you want to follow along, please install it like so:

```
$ go get github.com/pascaldekloe/jwt@v1.10.0
```

One of the first things that you need to consider when using JWTs is the choice of signing

algorithm.

If your JWT is going to be consumed by a *different* application to the one that created it, you should normally use an asymmetric-key algorithm like ECDSA or RSA. The ‘creating’ application uses its private key to sign the JWT, and the ‘consuming’ application uses the corresponding public key to verify the signature.

Whereas if your JWT is going to be *consumed by the same application that created it*, then the appropriate choice is a (simpler and faster) symmetric-key algorithm like HMAC-SHA256 with a random secret key. This is what we’ll use for our API.

So, to get authentication with JWTs working, you’ll first need to add a secret key for signing the JWTs to your `.envrc` file. For example:

File: `.envrc`

```
export GREENLIGHT_DB_DSN=postgres://greenlight:pa55word@localhost/greenlight
export JWT_SECRET=pei3einoh0Beem6uM6Ungohn2heiv5lah1ael4joopie5JaigeikoozaoTew2Eh6
```

Note: Your secret key should be a cryptographically secure random string with an underlying entropy of at least 32 bytes (256 bits).

And then you’ll need to update your `Makefile` to pass in the secret key as a command-line flag when starting the application, like so:

File: `Makefile`

```
...

# ===== #
# DEVELOPMENT
# ===== #

## run/api: run the cmd/api application
.PHONY: run/api
run/api:
    @go run ./cmd/api -db-dsn=${GREENLIGHT_DB_DSN} -jwt-secret=${JWT_SECRET}

...
```

Next you’ll need to edit the `cmd/api/main.go` file so that it parses the JWT secret from the command-line-flag into the `config` struct:

File: cmd/api/main.go

```
package main

...

type config struct {
    port int
    env string
    db struct {
        dsn string
        maxOpenConns int
        maxIdleConns int
        maxIdleTime string
    }
    limiter struct {
        enabled bool
        rps float64
        burst int
    }
    smtp struct {
        host string
        port int
        username string
        password string
        sender string
    }
    cors struct {
        trustedOrigins []string
    }
    jwt struct {
        secret string // Add a new field to store the JWT signing secret.
    }
}

...

func main() {
    var cfg config

    ...

    // Parse the JWT signing secret from the command-line-flag. Notice that we leave the
    // default value as the empty string if no flag is provided.
    flag.StringVar(&cfg.jwt.secret, "jwt-secret", "", "JWT secret")

    displayVersion := flag.Bool("version", false, "Display version and exit")

    flag.Parse()

    ...
}

...
```

And once that's done, you can change the `createAuthenticationTokenHandler()` so that it generates and sends a JWT instead of a stateful token. Like so:

File: cmd/api/tokens.go

```
package main
```

```

import (
    "errors"
    "net/http"
    "strconv" // New import
    "time"

    "greenlight.alexedwards.net/internal/data"
    "greenlight.alexedwards.net/internal/validator"

    "github.com/pascaldekloe/jwt" // New import
)

func (app *application) createAuthenticationTokenHandler(w http.ResponseWriter, r *http.Request) {
    var input struct {
        Email    string `json:"email"`
        Password string `json:"password"`
    }

    err := app.readJSON(w, r, &input)
    if err != nil {
        app.badRequestResponse(w, r, err)
        return
    }

    v := validator.New()

    data.ValidateEmail(v, input.Email)
    data.ValidatePasswordPlaintext(v, input.Password)

    if !v.Valid() {
        app.failedValidationResponse(w, r, v.Errors)
        return
    }

    user, err := app.models.Users.GetByEmail(input.Email)
    if err != nil {
        switch {
        case errors.Is(err, data.ErrRecordNotFound):
            app.invalidCredentialsResponse(w, r)
        default:
            app.serverErrorResponse(w, r, err)
        }
        return
    }

    match, err := user.Password.Matches(input.Password)
    if err != nil {
        app.serverErrorResponse(w, r, err)
        return
    }

    if !match {
        app.invalidCredentialsResponse(w, r)
        return
    }

    // Create a JWT claims struct containing the user ID as the subject, with an issued
    // time of now and validity window of the next 24 hours. We also set the issuer and
    // audience to a unique identifier for our application.
    var claims jwt.Claims
    claims.Subject = strconv.FormatInt(user.ID, 10)
    claims.Issued = jwt.NewNumericTime(time.Now())
    claims.NotBefore = jwt.NewNumericTime(time.Now())
    claims.Expires = jwt.NewNumericTime(time.Now().Add(24 * time.Hour))
    claims.Issuer = "greenlight.alexedwards.net"

```

```

claims.Audiences = []string{"greenlight.alexedwards.net"}

// Sign the JWT claims using the HMAC-SHA256 algorithm and the secret key from the
// application config. This returns a []byte slice containing the JWT as a base64-
// encoded string.
jwtBytes, err := claims.HMACSign(jwt.HS256, []byte(app.config.jwt.secret))
if err != nil {
    app.serverErrorResponse(w, r, err)
    return
}

// Convert the []byte slice to a string and return it in a JSON response.
err = app.writeJSON(w, http.StatusCreated, envelope{"authentication_token": string(jwtBytes)}, nil)
if err != nil {
    app.serverErrorResponse(w, r, err)
}
}
...

```

Go ahead and vendor the new github.com/pascaldekloe/jwt dependency and run the API like so:

```

$ make vendor
$ make run/api

```

Then when you make a request to the `POST /v1/tokens/authentication` endpoint with a valid email address and password, you should now get a response containing a JWT like this (line breaks added for readability):

```

$ curl -X POST -d '{"email": "faith@example.com", "password": "pa55word"}' localhost:4000/v1/tokens/authentication
{
  "authentication_token": "eyJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJncmVlbmV4Z2h0LmFsZXh1ZDh0cmRzLm5ldCIsInN1YiI6IjciLCJhdWQiOiZ3JlZW5saWdodC5hbGV4ZWR3YXJkcy5uZXQiXSwiZXhwIjoxNjE4OTM4MjY0Ljg0OTIwNSwiImIjoNjE4ODUxODY0Ljg0OTIwNSwiYW0IjoNjE4ODUxODY0Ljg0OTIwNDh9.zNK1bJPL5rLr_Yvjy0XuimJwaC3KgPqmW2M1u5RvgeA"
}

```

If you're curious, you can [decode the base64-encoded JWT data](#). You should see that the content of the claims matches the information that you would expect, similar to this:

```

{"alg": "HS256", "iss": "greenlight.alexedwards.net", "sub": "7", "aud": ["greenlight.alexedwards.net"],
"exp": 1618938264.819205, "nbf": 1618851864.819205, "iat": 1618851864.8192048}...

```

Next you'll need to update the `authenticate()` middleware to accept JWTs in an `Authorization: Bearer <jwt>` header, verify the JWT, and extract the user ID from the `subject` claim.

When we say “verify the JWT”, what we actually mean is the following four things:

- Check that the signature of the JWT matches the contents of the JWT, given our secret key. This will confirm that the token hasn't been altered by the client.
- Check that the current time is between the "not before" and "expires" times for the JWT.
- Check that the JWT "issuer" is "greenlight.alexedwards.net".
- Check that "greenlight.alexedwards.net" is in the JWT "audiences".

Go ahead and update the `authenticate()` middleware as follows:

File: cmd/api/middleware.go

```
package main

import (
    "errors"
    "expvar"
    "fmt"
    "net/http"
    "strconv"
    "strings"
    "sync"
    "time"

    "greenlight.alexedwards.net/internal/data"

    "github.com/felixge/httpsnoop"
    "github.com/pascaldekloe/jwt" // New import
    "github.com/tomasen/realip"
    "golang.org/x/time/rate"
)

...

func (app *application) authenticate(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        w.Header().Add("Vary", "Authorization")

        authorizationHeader := r.Header.Get("Authorization")

        if authorizationHeader == "" {
            r = app.contextSetUser(r, data.AnonymousUser)
            next.ServeHTTP(w, r)
            return
        }

        headerParts := strings.Split(authorizationHeader, " ")
        if len(headerParts) != 2 || headerParts[0] != "Bearer" {
            app.invalidAuthenticationTokenResponse(w, r)
            return
        }

        token := headerParts[1]

        // Parse the JWT and extract the claims. This will return an error if the JWT
        // contents doesn't match the signature (i.e. the token has been tampered with)
        // or the algorithm isn't valid.
        claims, err := jwt.HMACCheck([]byte(token), []byte(app.config.jwt.secret))
        if err != nil {
            app.invalidAuthenticationTokenResponse(w, r)
            return
        }
    })
}
```

```

// Check if the JWT is still valid at this moment in time.
if !claims.Valid(time.Now()) {
    app.invalidAuthenticationTokenResponse(w, r)
    return
}

// Check that the issuer is our application.
if claims.Issuer != "greenlight.alexedwards.net" {
    app.invalidAuthenticationTokenResponse(w, r)
    return
}

// Check that our application is in the expected audiences for the JWT.
if !claims.AcceptAudience("greenlight.alexedwards.net") {
    app.invalidAuthenticationTokenResponse(w, r)
    return
}

// At this point, we know that the JWT is all OK and we can trust the data in
// it. We extract the user ID from the claims subject and convert it from a
// string into an int64.
userID, err := strconv.ParseInt(claims.Subject, 10, 64)
if err != nil {
    app.serverErrorResponse(w, r, err)
    return
}

// Lookup the user record from the database.
user, err := app.models.Users.Get(userID)
if err != nil {
    switch {
    case errors.Is(err, data.ErrRecordNotFound):
        app.invalidAuthenticationTokenResponse(w, r)
    default:
        app.serverErrorResponse(w, r, err)
    }
    return
}

// Add the user record to the request context and continue as normal.
r = app.contextSetUser(r, user)

next.ServeHTTP(w, r)
})
}

```

Lastly, to get this working, you'll need to create a new `UserModel.Get()` method to retrieve the user details from the database based on their ID.

File: internal/data/users.go

```
package data

...

func (m UserModel) Get(id int64) (*User, error) {
    query := `
        SELECT id, created_at, name, email, password_hash, activated, version
        FROM users
        WHERE id = $1`

    var user User

    ctx, cancel := context.WithTimeout(context.Background(), 3*time.Second)
    defer cancel()

    err := m.DB.QueryRowContext(ctx, query, id).Scan(
        &user.ID,
        &user.CreatedAt,
        &user.Name,
        &user.Email,
        &user.Password.hash,
        &user.Activated,
        &user.Version,
    )

    if err != nil {
        switch {
        case errors.Is(err, sql.ErrNoRows):
            return nil, ErrRecordNotFound
        default:
            return nil, err
        }
    }

    return &user, nil
}
```

You should now be able to make a request to the one of the protected endpoints, and it will only succeed if your request contains a valid JWT in the **Authorization** header. For example:

```
$ curl -H "Authorization: Bearer eyJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJncmVlbmxpZ..." localhost:4000/v1/movies/2
{
  "movie": {
    "id": 2,
    "title": "Black Panther",
    "year": 2018,
    "runtime": "134 mins",
    "genres": [
      "sci-fi",
      "action",
      "adventure"
    ],
    "version": 2
  }
}

$ curl -H "Authorization: Bearer INVALID" localhost:4000/v1/movies/2
{
  "error": "invalid or missing authentication token"
}
```

Note: Because JWTs are base64-encoded, you may be able to change the final character in the JWT and it will still be accepted as valid. This is OK, and a detailed explanation of why can be found in [this StackOverflow post](#) and this [GitHub issue](#).

If you're planning to put a system into production which uses JWTs, I also recommend spending some time to read and fully understand the following two articles:

- [JWT Security Best Practices](#)
- [Critical vulnerabilities in JSON Web Token libraries](#)

JSON Encoding Nuances

Nil and empty slices are encoded differently

Nil slices in Go will be encoded to the `null` JSON value, whereas an empty (but not nil) slice will be encoded to an empty JSON array. For example:

```
var nilSlice []string
emptySlice := []string{}

m := map[string][]string{
    "nilSlice": nilSlice,
    "emptySlice": emptySlice,
}
```

Will encode to the following JSON:

```
{"emptySlice":[],"nilSlice":null}
```

Using `omitempty` on a zero-valued struct doesn't work

The `omitempty` struct tag directive *never considers a struct to be empty* — even if all the struct fields have their zero value and you use `omitempty` on those fields too. It will always appear as an object in the encoded JSON. For example, the following struct:

```
s := struct {
    Foo struct {
        Bar string `json:",omitempty"`
    } `json:",omitempty"`
}{}
```

Will encode to the JSON:

```
{"Foo":{}}
```

There's a long-standing [proposal](#) which discusses changing this behavior, but the Go 1 compatibility promise means that it's unlikely to happen any time soon. Instead, you can get around this by making the field *a pointer to a struct*, which works because `omitempty` considers nil pointers to be empty. For example:

```
s := struct {
    Foo *struct {
        Bar string `json:",omitempty"`
    } `json:",omitempty"`
}{}

```

Will encode to an empty JSON object like this:

```
{}
```

Using `omitempty` on a zero-value `time.Time` doesn't work

Using `omitempty` on a zero-value `time.Time` field won't hide it in the JSON object. This is because the `time.Time` type is a struct behind the scenes and, as mentioned above, `omitempty` never considers structs to be empty. Instead, the string `"0001-01-01T00:00:00Z"` will appear in the JSON (which is the value returned by calling the `MarshalJSON()` method on a zero-value `time.Time`). For example, the following struct:

```
s := struct {
    Foo time.Time `json:",omitempty"`
}{}

```

Will encode to the JSON:

```
{"Foo": "0001-01-01T00:00:00Z"}
```

Non-ASCII punctuation characters aren't supported in struct tags

When using struct tags to change the keys in a JSON object, any tags containing non-ASCII punctuation characters will be ignored. Notably this means that you can't use `en` or `em` dashes, or most currency signs, in struct tags. For example, the following struct:

```
s := struct {
    CostUSD string `json:"cost $"` // This is OK.
    CostEUR string `json:"cost €"` // This contains the non-ASCII punctuation character
                                   // € and will be ignored.
}{
    CostUSD: "100.00",
    CostEUR: "100.00",
}

```

Will encode to the following JSON (notice that the struct tag renaming the `CostEUR` key has been ignored):

```
{"cost $": "100.00", "CostEUR": "100.00"}
```

Integer, time.Time and net.IP values can be used as map keys

It's possible to encode a map which has integer values as the map keys. The integers will be automatically converted to strings in the resulting JSON (because the keys in a JSON object must always be strings). For example, the following map:

```
m := map[int]string{
    123: "foo",
    456_000: "bar",
}
```

Will encode to the JSON:

```
{"123": "foo", "456000": "bar"}
```

In addition, map keys that implement the [encoding.TextMarshaler](#) interface are also supported. This means that you can also use `time.Time` and `net.IP` values as map keys out-of-the-box. For example, the following map:

```
t1 := time.Now()
t2 := t1.Add(24 * time.Hour)

m := map[time.Time]string{
    t1: "foo",
    t2: "bar",
}
```

Will encode to JSON which looks similar to this:

```
{"2009-11-10T23:00:00Z": "foo", "2009-11-11T23:00:00Z": "bar"}
```

Angle brackets and ampersands in strings are escaped

If a string contains the angle brackets `<` or `>` these will be escaped to the [unicode character codes](#) `\u003c` and `\u003e` when encoded to JSON. Likewise the `&` character will be escaped to `\u0026`. This is to prevent some web browsers from accidentally interpreting a JSON response as HTML. For example, the following slice:

```
s := []string{
    "<foo>",
    "bar & baz",
}
```

Will encode to the JSON:

```
["\u003cfoo\u003e","bar \u0026 baz"]
```

If you want to prevent these characters being escaped, you'll need to use a `json.Encoder` instance with `SetEscapeHTML(false)` to perform the encoding.

Trailing zeroes are removed from floats

When encoding a floating-point number with a fractional part that ends in zero(es), any trailing zeroes will not appear in the JSON. For example:

```
s := []float64{
    123.0,
    456.100,
    789.990,
}
```

Will be encoded to the JSON:

```
[123,456.1,789.99]
```

Working with pre-computed JSON

If you have a string or `[]byte` slice which contains 'pre-computed' or 'pre-encoded' JSON, by default Go will treat it just like any other string or `[]byte` slice during encoding. That means that a string will be escaped and encoded as a JSON string, and a byte slice will be encoded as a base-64 JSON string. For example, the following struct:

```
m := struct {
    Person string
}{
    Person: `{"name": "Alice", "age": 21}`,
}
```

Will encode to:

```
{"Person":{"name": "\\"Alice\\", \\"age\\": 21}"}
```

If you want to interpolate the pre-computed JSON without any changes, you'll need to convert the pre-computed JSON value to a `json.RawMessage` type. Go will then directly interpolate it into the rest of the JSON. For example:

```
m := struct {  
    Person json.RawMessage  
}{  
    Person: json.RawMessage(`{"name": "Alice", "age": 21}`),  
}
```

Will encode to the JSON:

```
{"Person":{"name":"Alice","age":21}}
```

Important: Take care when using `json.RawMessage` to make sure that the pre-computed value contains valid JSON, otherwise you'll get an error when trying to encode the parent object. If you need to check this at runtime, you can do so by using the `json.Valid()` function.

The MarshalText fallback

If a type *doesn't* have a `MarshalJSON()` method but *does* have a `MarshalText()` method instead (so that it implements the `encoding.TextMarshaler` interface), then Go will fall back to calling this during JSON encoding and present the result as a JSON string.

For example, if you [run the following code](#):

```

type myFloat float64

func (f myFloat) MarshalText() ([]byte, error) {
    return []byte(fmt.Sprintf("%.2f", f)), nil
}

func main() {
    f := myFloat(1.0/3.0)

    js, err := json.Marshal(f)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Printf("%s", js)
}

```

It will print the return value from `MarshalText()` as a JSON string:

```
"0.33"
```

The receiver matters when using MarshalJSON

We touched on this earlier in the book, but it's really worth emphasizing because it often catches people out. If you implement a `MarshalJSON()` method on a custom type and the `MarshalJSON()` method uses a pointer receiver, it *will only be used when you are encoding a pointer to the custom type*. For any custom type *values*, it will be completely ignored.

You can see this in action if you run [this code on the Go playground](#).

Unless you specifically want this behavior, I recommend getting into the habit of always using value receivers for a `MarshalJSON()` method, just like we have in this book.

JSON Decoding Nuances

Decoding into Go arrays

When you're decoding a JSON array into a Go array (not a slice) there are a couple of important behaviors to be aware of:

- If the Go array is smaller than the JSON array, then the additional JSON array elements are silently discarded.
- If the Go array is larger than the JSON array, then the additional Go array elements are set to their zero values.

As an example:

```
js := `[1, 2, 3]`

var tooShortArray [2]int
err := json.NewDecoder(strings.NewReader(js)).Decode(&tooShortArray)
if err != nil {
    log.Fatal(err)
}

var tooLongArray [4]int
err = json.NewDecoder(strings.NewReader(js)).Decode(&tooLongArray)
if err != nil {
    log.Fatal(err)
}

fmt.Printf("tooShortArray: %v\n", tooShortArray)
fmt.Printf("tooLongArray: %v\n", tooLongArray)
```

Will print out:

```
tooShortArray: [1 2]
tooLongArray: [1 2 3 0]
```

Partial JSON decoding

If you have a lot of JSON input to process and only need a small part of it, it's often possible to leverage the `json.RawMessage` type to help deal with this. For example:

```
// Let's say that the only thing we're interested in is processing the "genres" array in
// the following JSON object
js := `{"title": "Top Gun", "genres": ["action", "romance"], "year": 1986}`

// Decode the JSON object to a map[string]json.RawMessage type. The json.RawMessage
// values in the map will retain their original, un-decoded, JSON values.
var m map[string]json.RawMessage

err := json.NewDecoder(strings.NewReader(js)).Decode(&m)
if err != nil {
    log.Fatal(err)
}

// We can then access the JSON "genres" value from the map and decode it as normal using
// the json.Unmarshal() function.
var genres []string

err = json.Unmarshal(m["genres"], &genres)
if err != nil {
    log.Fatal(err)
}

fmt.Printf("genres: %v\n", genres)
```

This will print out:

```
genres: [action romance]
```

In this toy example, using `json.RawMessage` doesn't save us much work. But if you need to process a JSON object with tens or hundreds of key/value pairs and only need a few of them, then taking this approach can save you a lot of typing.

Decoding into `interface{}` types

It's possible to decode JSON values into empty interface types (`interface{}`). When you do this, the underlying value that the `interface{}` type holds will depend on the type of the JSON value being decoded.

JSON type	⇒	Underlying Go type of interface{}
JSON boolean	⇒	<code>bool</code>
JSON string	⇒	<code>string</code>
JSON number	⇒	<code>float64</code>
JSON array	⇒	<code>[]interface{}</code>
JSON object	⇒	<code>map[string]interface{}</code>
JSON null	⇒	<code>nil</code>

Decoding into an `interface{}` type can be useful in situations where:

- You don't know in advance exactly what you're decoding.
- You need to decode JSON arrays which contain items with different JSON types.
- The key/value pair in a JSON object doesn't always contain values with the same JSON type.

As an example, consider the following code:

```
// This JSON array contains both JSON string and JSON boolean types.
js := `["foo", true]`

// Decode the JSON into a []interface{} slice.
var s []interface{}

err := json.NewDecoder(strings.NewReader(js)).Decode(&s)
if err != nil {
    log.Fatal(err)
}

// The first value in the slice will have the underlying Go type string, the second will
// have the underlying Go type bool. We can then type assert them and print them out
// the values along with their underlying type.
fmt.Printf("item: 0; type: %T; value: %v\n", s[0], s[0].(string))
fmt.Printf("item: 1; type: %T; value: %v\n", s[1], s[1].(bool))
```

This will print out:

```
item: 0; type: string; value: foo
item: 1; type: bool; value: true
```

Decoding a JSON number to an interface{}

As shown in the table above, when you decode a JSON number into an `interface{}` type

the value will have the underlying type `float64` — even if it is an integer in the original JSON. For example:

```
js := `10` // This JSON number is an integer.

var n interface{}

err := json.NewDecoder(strings.NewReader(js)).Decode(&n)
if err != nil {
    log.Fatal(err)
}

fmt.Printf("type: %T; value: %v\n", n, n)
```

Will print:

```
type: float64; value: 10
```

If you want to get the value as an integer (instead of a `float64`) you should call the `UseNumber()` method on your `json.Decoder` instance before decoding. This will cause all JSON numbers to be decoded to the underlying type `json.Number` instead of `float64`.

The `json.Number` type then provides an `Int64()` method that you can call to get the number as an `int64`, or the `String()` method to get the number as a `string`. For example:

```
js := `10`

var n interface{}

dec := json.NewDecoder(strings.NewReader(js))
dec.UseNumber() // Call the UseNumber() method on the decoder before using it.
err := dec.Decode(&n)
if err != nil {
    log.Fatal(err)
}

// Type assert the interface{} value to a json.Number, and then call the Int64() method
// to get the number as a Go int64.
nInt64, err := n.(json.Number).Int64()
if err != nil {
    log.Fatal(err)
}

// Likewise, you can use the String() method to get the number as a Go string.
nString := n.(json.Number).String()

fmt.Printf("type: %T; value: %v\n", n, n)
fmt.Printf("type: %T; value: %v\n", nInt64, nInt64)
fmt.Printf("type: %T; value: %v\n", nString, nString)
```

This will print out:

```
type: json.Number; value: 10
type: int64; value: 10
type: string; value: 10
```

Struct tag directives

Using the struct tag `json:"-"` on a struct field will cause it to be *ignored* when decoding JSON, even if the JSON input contains a corresponding key/value pair. For example:

```
js := `{"name": "alice", "age": 21}`

var person struct {
    Name string `json:"name"`
    Age  int32  `json:"- "`
}

err := json.NewDecoder(strings.NewReader(js)).Decode(&person)
if err != nil {
    log.Fatal(err)
}

fmt.Printf("%+v", person)
```

Will print out:

```
{Name:alice Age:0}
```

The `omitempty` struct tag directive does not have any effect on JSON decoding behavior.

Request Context Timeouts

As an alternative to the pattern that we’ve used in this book for managing database timeouts, we could have created a context with a timeout *in our handlers* using the *request context* as the parent — and then passed the context on to our database model.

Very roughly, a pattern like this:

```
func (app *application) exampleHandler(w http.ResponseWriter, r *http.Request) {
    ...

    // Create a context.Context with a one-second timeout deadline and which has the
    // request context as the 'parent'.
    ctx, cancel := context.WithTimeout(r.Context(), time.Second)
    defer cancel()

    // Pass the context on to the Get() method.
    example, err := app.models.Example.Get(ctx, id)
    if err != nil {
        switch {
        case errors.Is(err, data.ErrNotFound):
            app.notFoundResponse(w, r)
        default:
            app.serverErrorResponse(w, r, err)
        }
        return
    }
    ...
}
```

In many ways, doing this would be considered good practice. Allowing context to ‘flow’ through your application is generally a good idea, and this approach is also more flexible — whatever is calling the database model can easily control the timeout duration rather than it always being a fixed duration set by the database model.

But using the request context as the parent introduces a lot of additional complexity.

The key thing to be aware of is that the *request context will be canceled* if the client closes their HTTP connection. From the [net/http](#) docs:

For incoming server requests, the [request] context is canceled when the client’s connection closes, the request is canceled (with HTTP/2), or when the ServeHTTP method returns.

If we use the request context as the parent, then this cancellation signal will bubble down to our database driver `pq` and the running SQL query will be terminated in exactly the same

way that it is when a context timeout is reached. Importantly, it means we would receive the same `pq: canceling statement due to user request` error message that we do when a database query times out.

On one hand this is a positive thing — if there is no client left to return a response to, we may as well cancel the SQL query and free-up resources.

On the other hand, we will receive the same error message in two very different scenarios:

- When a database query takes too long to complete, in which case we would want to log it as an error.
- When a client closes the connection — which can happen for many innocuous reasons, such as a user closing a browser tab or terminating a process. It's not *really* an error from our application's point of view, and we would probably want to either ignore it or perhaps log it as a warning.

Fortunately, it is possible to tell these two scenarios apart by calling the `ctx.Err()` method on the context. If the context was canceled (due to a client closing the connection), then `ctx.Err()` will return the error `context.Canceled`. If the timeout was reached, then it will return `context.DeadlineExceeded` instead. If both the deadline is reached and the context is canceled, then `ctx.Err()` will surface whichever happened first.

It's also important to be aware that you may receive a `context.Canceled` error if the client closes the connection while the query is queued by `sql.DB`, and likewise `Scan()` may return a `context.Canceled` error too.

Putting all that together, a sensible way to manage this is to check for the error `pq: canceling statement due to user request` in our database model and wrap it with the error from `ctx.Err()` before returning. For example:

```

func (m ExampleModel) Get(ctx context.Context, id int64) (*Example, error) {
    query := `SELECT ... FROM examples WHERE id = $1`

    var example Example

    err := m.DB.QueryRowContext(ctx, query, id).Scan(...)

    if err != nil {
        switch {
        case err.Error() == "pq: canceling statement due to user request":
            // Wrap the error with ctx.Err().
            return nil, fmt.Errorf("%v: %w", err, ctx.Err())
        case errors.Is(err, sql.ErrNoRows):
            return nil, ErrNotFound
        default:
            return nil, err
        }
    }

    return &example, nil
}

```

Then in your handlers you can use `errors.Is()` to check if the error returned by the database model is equal to (or wraps) `context.Canceled`, and manage it accordingly.

```

func (app *application) exampleHandler(w http.ResponseWriter, r *http.Request) {
    ...

    ctx, cancel := context.WithTimeout(r.Context(), 3*time.Second)
    defer cancel()

    example, err := app.models.Example.Get(ctx, id)
    if err != nil {
        switch {
        // If the error is equal to or wraps context.Canceled, then return without taking
        // any further action.
        case errors.Is(err, context.Canceled):
            return
        case errors.Is(err, data.ErrNotFound):
            app.notFoundResponse(w, r)
        default:
            app.serverErrorResponse(w, r, err)
        }
        return
    }

    ...
}

```

As well as this additional error handling, you also need to be aware of the implications when using background goroutines. Bear in mind what I quoted earlier about request context cancellation:

For incoming server requests, the [request] context is canceled ... when the ServeHTTP method returns.

This means that if you derive your context from the request context, any SQL queries using the context in a background goroutine will be canceled when the HTTP response is sent for the initial request! If you don't want that to be the case (and you probably don't), then you would need to create a brand-new context for the background goroutine using `context.Background()` anyway.

So, to summarize, using the request context as the parent context for database timeouts adds quite a lot of behavioral complexity and introduces nuances that you and anyone else working on the codebase needs to be aware of. You have to ask: *is it worth it?*

For most applications, on most endpoints, it's probably not. The exceptions are probably applications which frequently run close to saturation point of their resources, or for specific endpoints which execute slow running or very computationally expensive SQL queries. In those cases, canceling queries aggressively when a client disappears may have a meaningful positive impact and make it worth the trade-off.

Feedback

Thanks for reading *Let's Go Further*. I hope that you've found it practical and informative, and that you'll enjoy using the knowledge and code patterns in your own projects 😊

I really appreciate any feedback (positive and negative), so that I can continue to improve the book in future updates. I've set up a short online survey at the URL below, and would be grateful if you could take a minute to fill it out:

› <https://forms.gle/FTXaziT7GLb3LccW6>

Cheers, and happy coding!

Alex