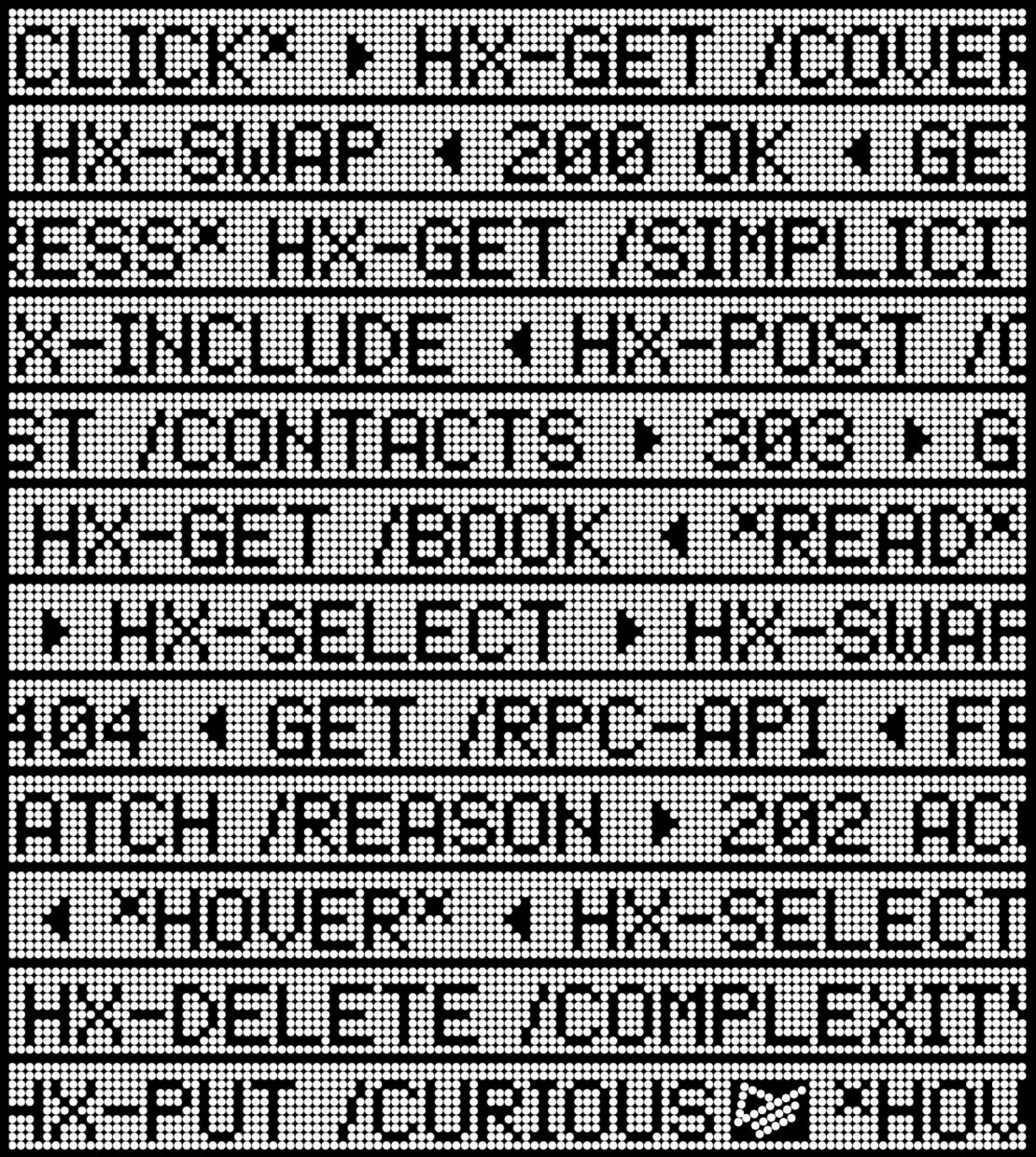
Hypermedia разработка HTMX и Hyperview

Карсон Гросс
Адам Степински
Денис Акшимшек



HYPERMEDIA SYSTEMS

Carson Gross
Adam Stepinski
Deniz Akcimcek

HYPERMEDIA-РАЗРАБОТКА HTMX И HYPERVIEW

Карсон Гросс
Адам Степински
Денис Акшимшек

ПИТЕР®

Санкт-Петербург • Москва • Минск

2024

ББК 32.988.02-018
УДК 004.738.5
Г88

Гросс Карсон, Степински Адам, Акшимшек Денис

Г88 Hypermedia-разработка. htmx и Hyperview. — СПб.: Питер, 2024. — 368 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-4096-1

Опытные программисты, выросшие вместе со Всемирной паутиной, не уделяли идеям гипермедиа особого внимания. А молодые веб-разработчики не знают ничего, кроме одностраничных приложений и фреймворков, используемых для их создания.

Устаревшая технология, подходящая только для создания документов со ссылками, текстом и графикой? Ничего подобного! В вашем распоряжении — эффективная технология для построения приложений.

Познакомьтесь с новыми инструментами — htmx и Hyperview, которые используют гипермедиа в качестве системной архитектуры. Научитесь строить сложные пользовательские интерфейсы с использованием гипермедиа как базовой технологии: на базе htmx для веб-приложений и на базе Hyperview для мобильных. А затем исследуйте прикладные современные подходы к построению веб-приложений, в которых эта архитектура используется.

Гипермедиа-управляемая архитектура подойдет не для каждого приложения, но повышенная гибкость и простота станут огромным преимуществом. Даже если этот подход не улучшит вашу программу, вам стоит понять его суть, сильные и слабые стороны и отличия от традиционно применяемой методики. Веб-среда росла быстрее, чем любая другая распределенная система в истории, и веб-разработчики должны уметь использовать сильные стороны базовых технологий, которые сделали возможным этот рост.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.988.02-018
УДК 004.738.5

ISBN 979-8394025143 англ.
ISBN 978-5-4461-4096-1

КРАТКОЕ СОДЕРЖАНИЕ

СОДЕРЖАНИЕ

ЗАКЛЮЧЕНИЕ

ОБ АВТОРАХ

Карсон Гросс — сеньор-разработчик с богатым 30-летним опытом создания как фронтенд-, так и бэкенд-приложений. Сооснователь и технический директор компании LeadDyno. Читает курсы в Университете Монтаны (MSU, Montana State University). Главная сфера интересов — языки программирования и веб-разработка на основе технологий гипермедиа.

Адам Степински — директор по инженерным вопросам в компании Instawork. У него более 15 лет опыта разработки и масштабирования технологических платформ в различных компаниях, от стартапов до Google. Адам — создатель Hyperview, мобильной системы гипермедиа.

Денис Акшимшек — инженер-разработчик, увлекающийся гипермедиа и вопросами доступности. Работал в компаниях Rebase Ventures (Великобритания), Big Sky Software (США) и Commspace (ЮАР), где создавал фулстек-приложения, веб-сайты и скрипты.

ОТ ИЗДАТЕЛЬСТВА

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

ЧАСТЬ I

Концепции гипермедиа

ВВЕДЕНИЕ

Эта книга посвящена построению приложений с использованием систем гипермедиа. На первый взгляд выражение *«система гипермедиа»* (Hypermedia system) выглядит немного странно: как гипермедиа может быть *системой*? Разве это не обычный механизм связывания документов?

Как HTML для Всемирной паутины?

Что вообще означает *система* гипермедиа?

Да, HTML относится к механизмам гипермедиа. Однако в основе Всемирной паутины лежит не только HTML: протокол HTTP (Hyper Text Transfer Protocol) используется для передачи HTML от серверов к клиентам, и с ним связаны многие технические подробности и функции: кэширование, заголовки, коды ответа и т. д.

И конечно, существуют *серверы гипермедиа*, которые предоставляют клиентам *API гипермедиа* (да, API) по сети.

Наконец, нельзя забывать о крайне важном *клиенте гипермедиа*: программном клиенте, который способен выводить понятный человеку *гипермедийный ответ*, чтобы человек мог взаимодействовать с удаленной системой. Несомненно, самыми широко известными и часто используемыми клиентами гипермедиа являются веб-браузеры.

Пожалуй, веб-браузеры — самые сложные программы, которыми мы пользуемся. Они не только поддерживают HTML, CSS и многие другие форматы файлов, но и предоставляют исполнительную и программную среду JavaScript — настолько мощную, что веб-разработчики могут создавать в ней целые приложения, почти не уступающие по сложности *толстым клиентам* (то есть нативным приложениям).

Исполнительная среда JavaScript настолько мощная, что многие разработчики в наши дни игнорируют гипермедийные возможности браузера, предпочитая строить веб-приложения полностью на JavaScript. Такие приложения часто называют *одностраничными приложениями*, или *SPA* (Single Page Applications). Вместо того чтобы переходить между страницами, эти веб-приложения используют JavaScript для прямого обновления пользовательского интерфейса. Взаи-

модействие с сервером в этих приложениях обычно осуществляется с помощью вызовов JSON API через AJAX. А для обновления пользовательского интерфейса часто применяются интерфейсные библиотеки JavaScript в «реактивном» стиле.

В таких приложениях HTML становится довольно громоздким языком описания графического интерфейса, который используется просто потому, что «исторически» находится под рукой (точнее, в браузере).

Приложения, созданные в этом стиле, не являются *гипермедиа-управляемыми*; они не используют преимущества нижележащей гипермедиа-системы веб-технологий.

Чтобы объяснить, как выглядит гипермедиа-управляемое приложение, и сравнить его с популярным в наше время SPA-подходом, сначала необходимо исследовать всю *гипермедиа-систему* интернета, не ограничиваясь HTML. Необходимо рассмотреть *сетевую архитектуру* интернета, включая то, как веб-серверы предоставляют API гипермедиа и как эффективно использовать средства гипермедиа, доступные в гипермедийном *клиенте* (например, браузере).

Все вышеперечисленные компоненты важны для построения эффективных гипермедиа-управляемых приложений и составляют в совокупности *систему*, которая и делает гипермедиа столь мощной архитектурой.

Что такое система гипермедиа?

Чтобы понять, что представляет собой система гипермедиа, сначала разберем каноническую систему: Всемирную паутину. Рой Филдинг (Roy Fielding) — инженер, который помогал создавать спецификации и строить реализации многих первых составляющих веб-среды, — ввел термин *REST* (REpresentational State Transfer), то есть «передача состояния представления». В своей диссертации он описал REST как *сетевую архитектуру*, которая противопоставлялась более ранним способам построения распределенных программных систем.

Мы определяем *систему гипермедиа* как RESTful-систему, то есть систему, которая соответствует принципам сетевой архитектуры REST в исходном понимании этого термина Филдингом.

К сожалению, в наши дни термин REST обычно ассоциируется с JSON API, так как именно в этом контексте он обычно применяется в отрасли. Такое использование термина REST не совсем корректно, потому что JSON не является *естественным* гипермедийным языком из-за отсутствия средств управления гипермедиа. Обмен гипермедийной информацией — явное условие того, чтобы

система могла считаться RESTful. Как мы пришли к этой ситуации со столь некорректным использованием термина REST? Это долгая история, и подробнее мы поговорим об этом позже. А пока, если вам кажется, что REST обязательно подразумевает JSON, попробуйте отойти от этого представления на время чтения книги и взглянуть на концепцию по-новому.

Важно понимать, что в своей диссертации Филдинг описывал Всемирную паутину в том состоянии, в каком она существовала в конце 1990-х годов. Тогда она просто состояла из веб-браузеров, обменивавшихся гипермедиа. Эта система с ее простыми ссылками и формами была тем, что Филдинг называл RESTful.

До того, как JSON API станет популярным инструментом веб-разработки, оставалось еще десять лет; концепция REST относилась к гипермедиа и версии 1.0 веб-среды.

Гипермедиа-управляемые приложения

В этой книге мы взглянем на гипермедиа как на *системную архитектуру*, а затем исследуем некоторые практические, *современные* подходы к построению веб-приложений, в которых эта архитектура используется. Приложения, построенные в этом стиле, будем называть *гипермедиа-управляемыми приложениями*, или *HDA* (Hypermedia-Driven Applications); мы сравним их с популярным стилем, применяемым в наши дни, — одностраничными приложениями (SPA).

Гипермедиа-управляемые приложения строятся на основе системы гипермедиа, которая соблюдает гипермедийную функциональность нижележащей системы и использует ее.

Цель книги

Цель этой книги — понятно объяснить читателю, чем архитектура RESTful-системы гипермедиа *отличается* от других систем «клиент — сервер» и какие сильные (и слабые) стороны присущи гипермедийным решениям. Кроме того, мы надеемся убедить читателя, что архитектура гипермедиа *актуальна* в современной веб-разработке.

Мы рассмотрим инструменты, которые помогут оценить требования к приложениям и ответить на вопрос: «Можно ли построить это приложение как гипермедиа-управляемое?»

Надеемся, вы ответите «да» применительно ко многим приложениям.

Структура книги

Книга включает три части:

- Введение в гипермедиа, в котором особое внимание уделяется темам HTML и HTTP. Обзор основных концепций гипермедиа завершается созданием простого приложения для управления контактами «в стиле Web 1.0», Contact.app.
- Затем вы научитесь использовать `htmx`[1] — гипермедиа-ориентированную библиотеку JavaScript, разработанную авторами этой книги, и с ее помощью улучшите приложение Contact.app. Благодаря `htmx` мы добьемся уровня интерактивности, который многие разработчики посчитали бы невозможным без большой, сложной интерфейсной библиотеки, такой как React. С `htmx` мы сделаем это, используя гипермедиа в качестве системной архитектуры.
- Наконец, мы рассмотрим Hyperview — совершенно особую *мобильную* систему гипермедиа, которая имеет отношение к веб-технологиям, но отличается от них. Эту систему создал один из авторов этой книги, Адам Степински. Она поддерживает *специфические мобильные* возможности за счет предоставления не только мобильных гипермедиа-технологий, но и мобильного гипермедиа-клиента. Эти новые компоненты в сочетании с любым сервером HTTP позволяют создавать мобильные гипермедиа-управляемые приложения.

Заметим, что каждую часть книги можно изучать отдельно от других. Если вы уже хорошо разбираетесь в теме гипермедиа и знаете, как работают базовые приложения Web 1.0, вы можете переходить сразу к части II с описанием `htmx` и построения современных веб-приложений на основе технологий гипермедиа. Точно так же, если вы уже разбираетесь в `htmx` и хотите глубже изучить новое направление мобильных гипермедиа-технологий, можете переходить прямо к части, посвященной Hyperview.

Тем не менее книга создавалась с расчетом на последовательное изучение, и в разделах, посвященных `htmx` и Hyperview, используется приложение Web 1.0, описанное в конце части I.

Более того, даже если вы хорошо разбираетесь во всех концепциях гипермедиа и деталях HTML и HTTP, мы рекомендуем хотя бы бегло просмотреть несколько первых глав, чтобы освежить эти темы в памяти.

[1] https://htmx.org/

Гипермедиа: новое поколение

В наши дни гипермедиа нечасто является предметом обсуждения. Даже многие опытные программисты, выросшие вместе с Всемирной паутиной в конце 1990-х и начале 2000-х годов, не уделяли идеям гипермедиа особого внимания. Некоторые молодые веб-разработчики не знают ничего, кроме одностраничных приложений и фреймворков, используемых для их создания.

В частности, многие начинали свою карьеру с построения приложений React.js, взаимодействующих с сервером Node через JSON API; некоторые разработчики вообще никогда не рассматривали гипермедиа как систему.

Это весьма печальный факт — и откровенно говоря, не в последнюю очередь дело в неумении лидеров сообщества веб-разработки доносить информацию и объяснять преимущества гипермедийного подхода.

Идея гипермедиа была просто превосходной! И такой осталась!

К концу этой книги вы будете владеть инструментами и языком, который позволит вам эффективно применять эту отличную идею в своих приложениях. Кроме того, вы сможете распространить принципы и концепции систем гипермедиа в широком сообществе веб-разработчиков.

Гипермедиа *может* конкурировать с другими технологиями, гипермедиа *может* побеждать, архитектура гипермедиа *сможет* занять место одностраничных приложений — но *только* если умные люди (как вы) будут знать об этой концепции, применять ее при построении приложений и рассказывать о ней миру.

> “ Будущее не определено. Нет судьбы, кроме той, что мы творим сами.
>
> *Кайл Риз, «Терминатор 2: Судный день»*

Заметки об HTML: гипермедиа на практике

Очевидно, HTML играет центральную роль в истории, которую мы рассказываем. В конце каждой главы мы будем делиться тем, что узнали о написании HTML для гипермедиа-управляемых веб-приложений.

Для начала вспомним, что веб-приложения — не изолированные системы. Мы пишем HTML не только для конкретного приложения, но и для того, чтобы оно могло взаимодействовать с другими компонентами веб-инфраструктуры. Если мы будем писать код с прицелом на создание системы гипермедиа, то сможем более эффективно пользоваться всеми возможностями, доступными в Сети.

Язык HTML хорошо сочетается с гипермедиа, когда разметка пишется для всех составляющих системы гипермедиа. Он передает состояние приложения поль-

зователям, просматривающим сайты в браузере, а также тем, кто слушает экранный диктор, устно описывающий содержимое сайта. Он передает информацию о предназначении сайтов поисковым системам, которые собирают данные на программном уровне. Кроме того, он информирует о своем поведении других разработчиков максимально ясным способом.

Нет, хорошая разметка HTML не решит всех проблем. Мантра о том, что язык HTML «доступен по умолчанию», только мешает. Отказываясь от других технологий (например, JavaScript), мы лишимся важных возможностей. А чтобы убедиться, что все работает так, как задумано, все равно придется тестировать — везде и много.

Тем не менее хорошая разметка HTML послужит тому, что браузеры выполнят *бóльшую* часть работы за нас.

ГИПЕРМЕДИА: ПОВТОРНОЕ ЗНАКОМСТВО

В наши дни технология гипермедиа стала универсальной и почти так же распространена, как электричество.

Миллиарды людей ежедневно используют системы на основе гипермедиа, в основном на уровне взаимодействия с языком гипертекстовой разметки *HTML* (Hypertext Markup Language), передаваемом по протоколу *HTTP* (Hypertext Transfer Protocol); для этого используется веб-браузер, подключенный к Сети.

Такие системы используются для получения новостей, общения с друзьями, покупок в интернете, игр, отправки электронной почты и т. д. Разнообразие и количество онлайн-сервисов, предоставляемых через гипермедиа, потрясает воображение.

Тем не менее, несмотря на такую распространенность, тема гипермедиа сама по себе остается на удивление малоизученной; в основном ею занимаются специалисты. Да, можно без труда найти множество пособий о том, как писать HTML, создавать ссылки и формы и т. д. Однако HTML очень редко рассматривается как *технология гипермедиа*, а в более широком смысле и как комплекс систем гипермедиа.

Ситуация заметно отличается от той, что была на заре веб-разработки, когда такие концепции, как *REST* (Representational State Transfer) или *HATEOAS* (Hypermedia As The Engine of Application State, «гипермедиа как ядро состояния приложения»), часто обсуждались, уточнялись и вызывали споры между веб-разработчиками.

Как ни печально, сегодня многие ругают самый популярный язык гипермедиа — HTML: это громоздкий, устаревший язык разметки, который приходится применять в пользовательских интерфейсах веб-приложений, уже почти полностью создаваемых на JavaScript. Так получилось, что HTML присутствует в браузере и мы вынуждены с ним работать.

Такой подход удручает, и мы надеемся убедить вас в том, что гипермедиа — не просто устаревшая технология, которую приходится принимать и использовать.

Мы хотим показать, что гипермедиа — невероятно революционный, простой и *гибкий* способ построения надежных приложений: *гипермедиа-управляемых приложений*.

Хочется верить, что к концу этой книги вы, как и мы, почувствуете, что гипермедиа может стать хорошим выбором архитектуры приложения, которое вы планируете разрабатывать. Вариант гипермедиа-управляемого приложения на основе *системы гипермедиа* (такой, как веб) вполне жизнеспособен; более того, нередко это отличное решение для *современного* веб-приложения.

(И как будет показано в части, посвященной Hyperview, не только для веб-приложения.)

Что такое гипермедиа?

> Гипертекст — новая форма письма, выводимая на экран компьютера, которая ветвится или выполняет действие по запросу читателя. Гипертекст является нелинейной формой письма; он приносит практическую пользу только при выводе на экран.
>
> *Тед Нельсон,* https://archive.org/details/SelectedPapers1977/page/n7/mode/2up

Начнем с самого начала: что такое гипермедиа?

Гипермедиа — информационная среда (например, текст) с возможностью *нелинейного перехода* из одной своей точки в другую по встроенным гиперссылкам. Префикс «гипер-» происходит от греческого префикса «ὑπερ-», означающего «за пределами» или «сверх-». Он указывает, что гипермедиа выходит за рамки обычных, пассивно потребляемых информационных сред, таких как журналы или газеты.

Гиперссылки — классический пример того, что называется *элементом управления гипермедиа* (hypermedia control).

> *Элемент управления гипермедиа* — это элемент гипермедиа, описывающий некоторое взаимодействие (или управляющий этим взаимодействием), часто с удаленным сервером. Информация об этом взаимодействии напрямую и полностью кодируется внутри самого элемента.

Элементы управления — то, что отличает гипермедиа от других видов информационных средств.

Возможно, вам более знаком термин *«гипертекст»*; приведенное выше определение взято со страницы в «Википедии», посвященной гипертексту. Гипертекст составляет подкатегорию гипермедиа, и бóльшая часть этой книги будет по-

священа тому, как строить современные приложения на основе гипертекста, например HTML (Hypertext Markup Language) или HXML (гипертекста, используемого мобильной гипермедиа-системой Hyperview).

Гипертекст, как HTML, работает в сочетании с другими технологиями, необходимыми для функционирования всей системы гипермедиа: сетевыми протоколами (такими, как HTTP), другими типами информационных сред (например, графикой или видео), серверами гипермедиа (то есть серверами, предоставляющими API гипермедиа), полнофункциональными клиентами гипермедиа (например, веб-браузерами) и т. д.

По этой причине для описания базовой архитектуры приложений с применением гипертекста мы предпочитаем более широкий термин *системы гипермедиа*, чтобы подчеркнуть роль системной архитектуры в конкретной используемой разновидности гипермедиа.

Именно *архитектура* системы гипермедиа в целом не пользуется должным вниманием и игнорируется многими современными веб-разработчиками.

Краткая история гипермедиа

Как возникла идея гипермедиа?

Хотя у современной концепции гипертекста и более общей концепции гипермедиа было много предшественников, отправной точкой для формирования современных представлений о гипермедиа многие считают статью Ванневара Буша (Vannevar Bush) «As We May Think», вышедшую в 1945 году в журнале «Atlantic».

В этой статье Буш описал устройство, которое он назвал Memex. Это устройство, использовавшее сложную механическую систему катушек и микропленок в сочетании с системой кодирования, давало бы возможность пользователю переходить между взаимосвязанными кадрами контента. Устройство Memex так никогда и не было создано, но его идея послужила вдохновением для последующей проработки концепции гипермедиа.

Термины «гипертекст» и «гипермедиа» были введены в 1963 году Тедом Нельсоном (Ted Nelson), который работал над системой редактирования гипертекста в Университете Брауна, а позднее создал *FRESS* (File Retrieval and Editing System) — невероятно передовую для своего времени систему гипермедиа. (Не исключено, что это была первая цифровая система с возможностью «отмены» (undo).)

Пока Нельсон работал над своими идеями, Дуглас Энгельбарт (Douglas Engelbart) трудился в Стэнфордском исследовательском институте, пытаясь воплотить

в реальность устройство Memex Ванневара Буша. В 1968 году Энгельбарт провел «демонстрацию века» в Сан-Франциско, штат Калифорния.

Энгельбарт продемонстрировал ряд невероятных технологических достижений:

- дистанционное редактирование текста совместно с коллегами в Менло-Парке;
- видео- и аудиочат;
- интегрированную систему окон с возможностью изменения размеров окон и других параметров;
- гипертекст, в котором по щелчку на подчеркнутом фрагменте происходит переход к новому контенту.

Несмотря на бурные овации потрясенной после выступления аудитории, прошли десятилетия, прежде чем продемонстрированные Энгельбартом технологии получили массовое распространение.

Современная реализация

В 1990 году Тим Бернерс-Ли (Tim Berners-Lee), работавший в CERN, опубликовал первый веб-сайт. Он работал над идеей гипертекста около 10 лет. Наконец, в отчаянии от того факта, что ученым так трудно делиться результатами своих исследований, он нашел подходящий момент и организационную поддержку для создания Всемирной паутины:

> “Создание Всемирной паутины в действительности было актом отчаяния, потому что во время моей работы в CERN ситуация была очень сложной. Многие технологии, задействованные во Всемирной паутине (гипертекст, интернет, многошрифтовые текстовые объекты), уже были спроектированы. Мне оставалось лишь объединить их. Это было обобщением, переходом на более высокий уровень абстракции, попыткой рассматривать все системы документации как части более крупной воображаемой системы документации.
>
> *Тим Бернерс-Ли*
> https://britishheritage.org/tim-berners-lee-the-world-wide-web

К 1994 году его творение стало развиваться настолько быстро, что Бернерс-Ли основал W3C — группу компаний и исследователей, работавших над улучшением Паутины. Все стандарты, созданные W3C, не требовали лицензионных отчислений. Любой желающий мог адаптировать и реализовать их, что привело к укреплению открытой, ориентированной на сотрудничество природы Всемирной паутины.

В 2000 году Рой Филдинг, тогда работавший в Калифорнийском университете в Ирвайне, опубликовал во Всемирной паутине свою судьбоносную диссертацию «Architectural Styles and the Design of Network-based Software Architectures» («Архитектурные стили и дизайн программных архитектур сетевой среды»). Филдинг работал над HTTP-сервером с открытым исходным кодом Apache и в своей работе описывал то, что считал *новой особой сетевой архитектурой*, появившейся в ранней версии Паутины.

Филдинг работал над первыми спецификациями HTTP. В своей статье для определения сетевой модели гипермедиа он использовал термин REST (REpresentational State Transfer).

Работа Филдинга стала опорой для первых веб-разработчиков, получивших в свое распоряжение язык для обсуждения новой технической среды, в которой они строили приложения.

Мы подробно обсудим ключевые идеи Филдинга в главе 2, а заодно попытаемся навести порядок с определениями REST, HATEOAS и гипермедиа.

Самый успешный гипертекст: HTML

> “ В начале была гиперссылка, и гиперссылка была с веб-средой, и гиперссылка была веб-средой. И это было хорошо.
>
> *Rescuing REST From the API Winter*
> https://intercoolerjs.org/2016/01/18/rescuing-rest.html

HTML, над которым работали Бернерс-Ли, Филдинг и многие другие, создавался вокруг гипермедиа. Изначально разметка HTML представляла собой гипермедиа-текст, предназначенный только для чтения и используемый для публикации академических документов. Эти документы связывались при помощи якорных тегов, которые создавали между ними *гиперссылки*. При помощи этих ссылок пользователи могли быстро перемещаться от одного документа к другому.

В спецификации HTML 2.0 появилось понятие тега `form`, присоединившегося к якорному тегу (то есть гиперссылке) как второй элемент управления. Введение тега `form` открыло возможность построения *приложений* в веб-среде, так как оно предоставляло механизм *обновления* ресурсов, а не только их чтения.

В этот момент Всемирная паутина из интересной документоориентированной системы превратилась в перспективную *архитектуру приложений*.

В наши дни HTML является самой широко используемой технологией гипермедиа. Естественно, предполагается, что наш читатель достаточно хорошо знаком с ней. Вам не нужно быть экспертом в области HTML (или CSS), чтобы понять приведенный в книге код, но чем лучше вы разбираетесь в основных тегах и концепциях HTML, тем больше пользы принесет вам книга.

Сущность HTML как гипермедиа

Рассмотрим два определяющих гипермедиа-элемента (*элемента управления*) HTML — якорный тег и тег формы — более подробно.

Якорные теги

Якорные теги встречаются настолько часто, что кажутся набившими оскомину. И все же стоит рассмотреть механику работы гиперссылок, чтобы настроиться на более глубокое понимание гипермедиа.

Возьмем простой якорный тег, встроенный в документ HTML.

Листинг 1. Простая гиперссылка

```
<a href="https://hypermedia.systems/">
  Системы гипермедиа
 </a>
```

Якорный тег состоит из собственно тега `<a></a>`, а также атрибутов и контента внутри тега. Особый интерес представляет атрибут `href`, который задает *гипертекстовую ссылку* на другой документ или фрагмент документа. Именно этот атрибут превращает якорный тег в элемент гипермедиа.

Типичный браузер будет интерпретировать этот якорный тег следующим образом:

- Вывести текст «Системы гипермедиа» в такой форме, чтобы пользователь видел, что на нем можно щелкнуть.
- Когда пользователь щелкает на этот текст, выдать запрос `HTTP GET` к URL https://hypermedia.systems/.
- Принять HTML-контент из тела ответа HTTP на этот запрос и заменить весь экран в браузере новым документом, с обновлением адресной строки новым URL-адресом.

Якорные ссылки обеспечивают основной механизм перемещения в современной веб-среде — переход по ссылке от документа к документу или от ресурса к ресурсу.

Схематично взаимодействие пользователя с якорным тегом/гиперссылкой выглядит так:

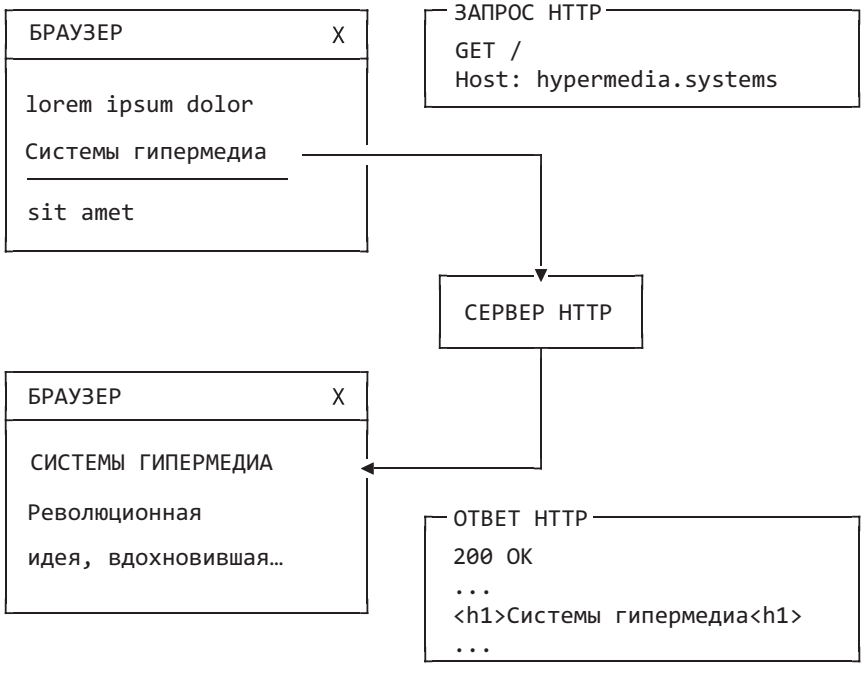


Рис. 1. Запрос HTTP GET в действии

При щелчке на ссылке браузер (или, как иногда говорят, *клиент гипермедиа*) инициирует `GET`-запрос HTTP к URL-адресу, закодированному в атрибуте `href` ссылки.

Обратите внимание, что запрос HTTP включает дополнительные данные (*метаданные*). Они описывают, что именно браузеру нужно получить от сервера, в форме заголовков. Эти заголовки и HTTP более подробно рассматриваются в главе 2.

Затем с *сервера гипермедиа* поступает *гипермедийный ответ* на запрос — разметка HTML — для новой страницы. На первый взгляд этот момент может показаться незначащим и очевидным, но это исключительно важный признак *системы гипермедиа*, действительно отвечающей условиям REST: клиент и сервер должны взаимодействовать через гипермедиа!

Теги форм

Якорные теги обеспечивают *навигацию* между документами или ресурсами, но не позволяют обновлять эти ресурсы. За эту функциональность отвечает тег `form`.

Простой пример формы HTML:

Листинг 2. Простая форма

```
<form action="/signup" method="post">
 <input type="text" name="email" placeholder="Укажите адрес электронной почты,
                          чтобы зарегистрироваться..."/>
 <button>Регистрация</button>
</form>
```

Как и якорный тег, тег формы состоит из собственно тега `<form></form>`, а также атрибутов и контента внутри тега. Обратите внимание: у тега формы нет атрибута `href`, но есть атрибут `action`, указывающий, куда следует направить запрос HTTP. Кроме того, он содержит атрибут `method`, который точно указывает, какой «метод» HTTP должен использоваться. В приведенном примере форма дает команду браузеру выдать запрос `POST`.

В отличие от якорных тегов, контент и теги *внутри* формы могут влиять на гипермедиа-взаимодействие между формой и сервером. *Значения* тегов `input` и других тегов (например, `select`) при отправке данных формы будут включены в запрос HTTP как параметры URL в случае `GET` и как часть тела запроса в случае `POST`. В отличие от якорного тега, это позволяет форме включать в запрос произвольный объем информации, полученной от пользователя.

Типичный браузер будет интерпретировать этот тег формы и его содержимое примерно следующим образом:

- Вывести текстовое поле ввода и кнопку «Регистрация» для пользователя.
- Когда пользователь отправит данные формы, щелкнув на кнопке «Регистрация» или нажав клавишу Enter, пока элемент `input` обладает фокусом, направить запрос HTTP `POST` по пути `/signup` «текущего» сервера.
- Принять HTML-контент из тела ответа HTTP и заменить весь экран в браузере новым документом, с обновлением адресной строки новым URL-адресом.

Этот механизм позволяет пользователю выдавать запросы на *обновление состояния* ресурсов на сервере. Заметим, что, несмотря на новый тип запроса, весь обмен данными между клиентом и сервером все еще происходит исключительно через *гипермедиа*.

Именно тег формы делает возможным существование гипермедиа-управляемых приложений.

Вероятно, опытный веб-разработчик заметит, что мы опустили некоторые подробности и трудные случаи. Например, ответ на отправку формы часто *перенаправляет* клиента на другой URL-адрес.

Это правда, и мы еще займемся техническими подробностями форм в следующих главах, а пока и этого простого примера будет достаточно для демонстрации базового механизма обновления состояния системы исключительно через гипермедиа.

Диаграмма взаимодействия выглядит так:

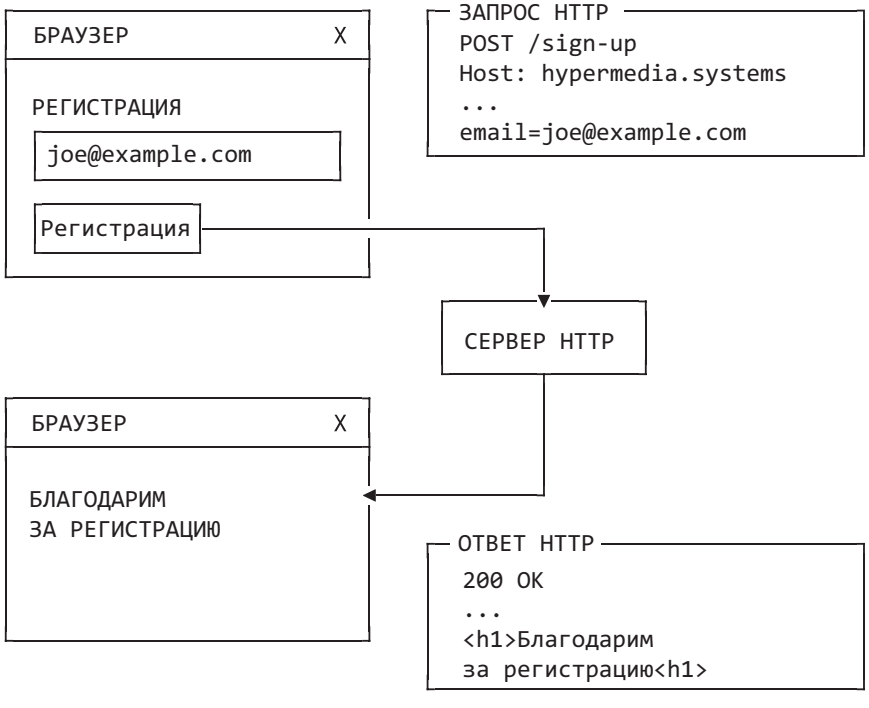


Рис. 2. Запрос HTTP POST в действии

Приложения Web 1.0

Как человеку, интересующемуся веб-разработкой, вам, вероятно, уже знакомы приведенные выше диаграммы и рассуждения. Возможно, кому-то этот материал даже покажется скучным. Но сделайте шаг назад и обратите внимание на тот факт, что этими двумя элементами гипермедиа — якорями и формами — ограничиваются *все* собственные возможности базового HTML для взаимодействия пользователя с сервером.

Всего два тега!

Тем не менее они обеспечили Всемирной паутине на заре ее развития рост с экспоненциальной скоростью и возможность предоставлять невероятно огромный объем динамической онлайн-функциональности миллиардам людей.

Это убедительное доказательство мощи гипермедиа. Даже сегодня, когда в веб-разработке все чаще на ведущие позиции выходят большие JavaScript-ориентированные интерфейсные фреймворки, многие предпочитают использовать для достижения своих целей простой базовый HTML, и нередко результат их полностью устраивает.

Эти два тега наделяют HTML колоссальной выразительностью.

Что не относится к гипермедиа?

Итак, ссылки и формы — два основных механизма гипермедиа, предназначенные для взаимодействия с сервером и доступные в HTML.

Теперь рассмотрим другой подход: взаимодействие с сервером посредством выдачи запроса HTTP через JavaScript. Для этого мы будем использовать `fetch()`[1] — популярный API для выдачи запросов «асинхронного JavaScript и XML», или запросов *AJAX* (*Asynchronous JavaScript and XML*), доступный во всех современных браузерах.

Листинг 3. JavaScript

```
<button onclick="fetch('/api/v1/contacts/1') ❶
        .then(response => response.json()) ❷
        .then(data => updateUI(data))">  ❸
  Fetch Contact
</button>
```

❶ Выдача запроса.

❷ Преобразование ответа в объект JavaScript.

❸ Вызов функции `updateUI()` с объектом.

Кнопка имеет атрибут `onclick`, в котором хранится код JavaScript, выполняемый при щелчке по ней.

JavaScript выдает запрос AJAX HTTP `GET` к пути `/api/v1/contacts/1` с использованием `fetch()`. Запрос AJAX похож на «обычный» запрос HTTP, но он выдается браузером скрыто. Пользователь не видит индикатор запроса от браузера, как в случае с обычными ссылками и формами. Кроме того, в отличие от запросов, выдаваемых этими элементами управления, за обработку ответа от сервера отвечает код JavaScript.

Хотя в акроним AJAX входит XML, в наши дни ответ HTTP на такой запрос почти наверняка будет закодирован в формате JSON (JavaScript Object Notation), а не в XML.

[1] https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

Ответ HTTP на этот запрос может выглядеть примерно так:

Листинг 4. JSON

```
{ ❶
 "id": 42, ❷
 "email" : "json-example@example.org" ❸
}
```

❶ Начало объекта JSON.
❷ Свойство — в данном случае с именем id и значением 42.
❸ Другое свойство — адрес электронной почты контакта с этим id.

Приведенный выше код JavaScript преобразует текст JSON, полученный от сервера, в объект JavaScript; для этого текст передается в вызове метода `json()`. Далее новый объект JavaScript передается методу `updateUI()`.

Метод `updateUI()` отвечает за обновление пользовательского интерфейса на основании данных, закодированных в объекте JavaScript, — возможно, с выводом данных контакта в разметке HTML, генерируемой клиентским шаблоном в приложении JavaScript.

Подробности того, что делает функция `updateUI()`, нам неважны.

А что важно и что является *критической* особенностью взаимодействий с сервером на основе JSON — то, что в них не используются гипермедиа. JSON API не возвращает гипермедийный ответ. В нем нет гиперссылок или других элементов гипермедиа. Здесь JSON API скорее является *API данных*.

Так как ответ закодирован в формате JSON, а *не* в формате гипермедиа, методу JavaScript `updateUI()` нужна информация, как преобразовать эти контактные данные в HTML.

В частности, коду `updateUI()` необходимы сведения о *внутренней структуре* и смысле данных. Он должен знать:

- точную структуру и имена полей в объекте данных JSON;
- какими отношениями они связаны;
- как обновить локальные данные, которым соответствуют эти новые данные;
- как отобразить эти данные в браузере;
- какие дополнительные действия / конечные точки API могут вызываться с этими данными.

Вкратце, логика `updateUI()` должна обладать полной информацией о конечной точке API, имеющей путь `/api/v1/contact/1`, причем эта информация должна предоставляться сторонним источником не внутри самого ответа. В результате

между кодом `updateUI()` и API возникают особые отношения, называемые *сильной связанностью* (tight coupling): если формат ответа JSON изменится, то код `updateUI()` почти наверняка тоже придется изменять.

Одностраничные приложения

Хотя приведенный фрагмент кода JavaScript выглядит очень скромно, это органическое начало целой большой концепции создания веб-приложений. С него начинаются *одностраничные приложения* (Single Page Application, SPA). Веб-приложение уже не осуществляет навигацию *между* страницами с использованием гипермедийных элементов управления, как это было со ссылками и формами.

Вместо этого приложение обменивается *обычными данными* с сервером, а затем обновляет контент *в пределах* одной страницы.

Когда такая стратегия или архитектура принимается для всего приложения, все фактически происходит на одной странице и приложение становится «одностраничным».

Архитектура одностраничных приложений стала чрезвычайно популярна и в последнее десятилетие доминирует при построении веб-приложений. Об этом говорят высокий уровень внимания отрасли к этой архитектуре и частота ее обсуждений.

В наши дни в подавляющем большинстве одностраничных приложений для управления пользовательским интерфейсом применяются намного более сложные фреймворки, чем показано в этом простом примере. Такие популярные библиотеки, как React, Angular, Vue.js и т. д., стали распространенным — и можно сказать, стандартным — способом построения веб-приложений.

С такими сложными фреймворками разработчики обычно применяют более тщательно проработанную модель на стороне клиента, а именно объекты JavaScript, хранимые локально в памяти браузера и представляющие «модель» или «предметную область» приложения. Эти объекты JavaScript обновляются кодом JavaScript, а фреймворк «реагирует» на эти изменения обновлением пользовательского интерфейса.

Когда пользовательский интерфейс (UI) обновляется пользователем, эти изменения также распространяются в объекты модели, устанавливая «двусторонний» механизм связывания: модель может обновить пользовательский интерфейс, а пользовательский интерфейс может обновить модель.

Это намного более сложный подход к веб-клиенту, чем гипермедиа. Обычно он почти полностью упраздняет нижележащую инфраструктуру гипермедиа, доступную в браузере.

HTML все еще используется для построения пользовательских интерфейсов, но *гипермедийный* аспект двух основных элементов гипермедиа, якорей и форм, остается незадействованным. Ни один из этих тегов не взаимодействует с сервером через нативный механизм *гипермедиа*. Вместо этого они становятся элементами пользовательского интерфейса, которые управляют локальными взаимодействиями с моделью предметной области в памяти через JavaScript; далее они синхронизируются с сервером с использованием простых данных JSON API.

Итак, как в приведенном выше примере с простой кнопкой, одностраничные приложения работают в обход архитектуры гипермедиа. При этом они отказываются от преимуществ существующей RESTful-архитектуры веб-среды и встроенной функциональности нативных элементов гипермедиа в пользу поведения, управляемого JavaScript.

SPA-приложения представляют собой нечто намного большее, чем *толстые клиенты*, то есть приложения «клиент — сервер» 1980-х — архитектура, которая была популярна *до* появления Всемирной паутины и реакцией на которую во многих отношениях и стало возникновение Всемирной паутины.

Конечно, такой подход не обязательно ошибочен: в некоторых случаях толстый клиент — вполне подходящий выбор для приложения. Однако стоит задуматься, *почему* веб-разработчики так часто выбирают этот вариант, не учитывая альтернативы, и нет ли причин выбрать другой подход.

Почему стоит использовать гипермедиа?

> “ Одностраничные приложения React с рендерингом на стороне сервера становятся своеобразной нормой веб-разработки. Два ключевых элемента этой архитектуры выглядят примерно так:
>
> 1. Основной пользовательский интерфейс строится и обновляется в JavaScript с применением React или похожей библиотеки.
> 2. Серверная часть представлена API, к которому приложение обращается с запросами.
>
> Эта идея действительно захватила интернет. Все началось с нескольких крупных популярных сайтов и дошло до маркетинговых сайтов и блогов.
>
> *Том Макрайт (Tom MacWright)*,
> https://macwright.com/2020/05/10/spa-fatigue.html

Схема одностраничных приложений на основе JavaScript захватила веб-разработку, и если бы кому-то предложили назвать единственную причину ее

бешеного успеха, он бы выбрал следующую: одностраничные приложения обеспечивают намного больший уровень интерактивности и иммерсивности, чем старые громоздкие приложения Web 1.0 на основе гипермедиа. SPA-приложения обладают целым рядом преимуществ: плавным обновлением элементов, встроенных в страницу, без перезагрузки всего документа; использованием переходов CSS для создания привлекательных визуальных эффектов; возможностью хука произвольных событий (например, перемещений мыши).

Все эти возможности обеспечивают приложениям на основе JavaScript огромное преимущество при построении сложных пользовательских интерфейсов.

С учетом популярности, мощи и успеха этого современного подхода к созданию веб-приложений зачем кому-то использовать старый, громоздкий и менее популярный подход, такой как гипермедиа?

Усталость от JavaScript

Хорошо, что вы спросили!

Оказывается, архитектура гипермедиа, даже в своей исходной форме Web 1.0, обладает рядом преимуществ по сравнению со связкой «SPA + JSON Data API». Вот три основных:

- Это чрезвычайно *простой* подход к построению веб-приложений.
- Он очень хорошо переносит изменения контента и API. Более того, он от них выигрывает!
- В нем используются проверенные, надежные механизмы веб-браузеров, такие как кэширование.

В частности, первые два преимущества решают серьезные проблемы современной веб-разработки:

- Одностраничная инфраструктура стала в высшей степени сложной, для управления ею часто требуются усилия всей команды.
- Пересмотр JSON API — постоянные изменения, вносимые в JSON API для поддержки потребностей приложения, — стал болевой точкой для многих команд разработки.

Наличие этих проблем в сочетании с другими проблемами (например, пересмотром библиотеки JavaScript) привело к явлению, известному как «усталость от JavaScript». Под ним понимается общее нежелание продираться через все препоны, чтобы хоть что-нибудь сделать в современных веб-приложениях.

Мы уверены, что архитектура гипермедиа способна излечить многие команды и отдельных разработчиков от «усталости от JavaScript».

Но если концепция гипермедиа настолько хороша и если она решает столько проблем, преследующих отрасль веб-разработки, почему она оказалась на обочине? В конце концов, она появилась раньше других. Почему веб-разработчики ей не следовали?

Существуют две основные причины, по которым решения гипермедиа не пользуются особой популярностью у веб-разработчиков.

Первая: выразительность HTML *как гипермедиа* практически не изменилась (или вообще не изменилась) с момента возникновения HTML 2.0 *в середине 1990-х*. Конечно, в HTML появилось много новых *функций*, но почти три десятилетия мы не видели *никаких* действительно новых способов взаимодействия с сервером из HTML.

Из элементов гипермедиа у разработчиков HTML по-прежнему только якорные теги и формы, и эти элементы управления могут выдавать только запросы `GET` и `POST`.

Необъяснимое отсутствие прогресса в HTML немедленно ведет ко второй и, пожалуй, более практической причине, по которой концепция «HTML как гипермедиа» не прижилась: пока интерактивность и выразительность HTML оставались на прежнем уровне, потребности веб-пользователей продолжали расти — пользователи ожидали все более и более интерактивных веб-приложений.

Приложения на основе JavaScript в сочетании с JSON API, ориентированными на данные, предложили способ реализации более сложных пользовательских интерфейсов. Именно опыт *взаимодействия с пользователем*, который можно было обеспечить в JavaScript и которого нельзя было достичь в простом HTML, подтолкнул сообщество веб-разработки к одностраничной архитектуре на основе JavaScript. Сдвиг не был вызван каким-то внутренним превосходством одностраничного приложения как системной архитектуры.

Но события не обязательно должны были развиваться по этому сценарию. В идею гипермедиа *не заложено* ничего, что помешало бы ей иметь более богатую и выразительную модель интерактивности, чем минимальный HTML. Вместо того чтобы уходить от решений на основе гипермедиа, отрасль могла бы потребовать большей интерактивности от HTML.

Вместо этого стандартом стало создание приложений в стиле толстого клиента для веб-браузеров — понятный переход к более знакомой модели для построения полнофункциональных приложений.

Конечно, не все отказались от идеи гипермедиа. Известны героические попытки продолжить продвижение гипермедиа за рамками HTML, такие как HyTime, VoiceXML и HAL.

Однако язык HTML — наиболее широко используемая среда гипермедиа — перестал развиваться как гипермедиа. Веб-разработка эволюционировала, решая

проблемы интерактивности в HTML переходом на модель SPA, основанную на JavaScript, и, в основном непреднамеренно, на совершенно иную системную архитектуру.

Возрождение гипермедиа?

Интересно представить, по какому пути *мог бы* пойти язык HTML. Вместо того чтобы застыть как гипермедийная среда, мог ли HTML продолжить свое развитие? Мог ли он дополняться новыми элементами гипермедиа и расширять выразительность уже существующих? Получилось бы создать современное веб-приложение в исходной гипермедиа-ориентированной RESTful-модели, которая сделала раннюю версию Всемирной паутины столь мощной, гибкой и увлекательной?

Кто-то скажет, что это всего лишь досужие домыслы, но у нас есть хорошие новости: за последнее десятилетие появились неординарные альтернативные интерфейсные библиотеки, которые пытаются вдохнуть новую жизнь в HTML. Как ни парадоксально, эти библиотеки написаны на JavaScript — технологии, заместившей HTML в веб-разработке.

Тем не менее эти библиотеки используют JavaScript не как *замену* фундаментальной гипермедийной системы веб-среды. JavaScript применяется для дополнения самого языка HTML *как платформы гипермедиа*.

Эти *гипермедиа-ориентированные* библиотеки снова делают гипермедиа основополагающей технологией веб-приложений.

Гипермедиа-ориентированные библиотеки JavaScript

В веб-разработке уже давно идут споры между сторонниками одностраничного (SPA) подхода и того, что сейчас называется «многостраничным» подходом, или MPA (Multi-Page Application). MPA — это просто новое обозначение старого способа построения веб-приложений в стиле Web 1.0 с использованием ссылок и форм, размещенных на разных веб-страницах, с отправкой запросов HTTP и получением ответов HTML.

Приложения MPA по своей природе являются гипермедиа-управляемыми; в конце концов, именно их Рой Филдинг описывал в своей диссертации.

Эти приложения кажутся громоздкими, но работают достаточно хорошо. Многие веб-разработчики и команды решают выбрать ограничения простого HTML в пользу простоты и надежности.

Рич Харрис (Rich Harris), создатель Svelte.js (популярной библиотеки SPA) и идейный лидер стороны SPA в этом споре, предложил объединить старый стиль MPA с новым стилем SPA. Харрис называет такой подход к построению

веб-приложений «переходным» в том смысле, что он пытается осуществить слияние MPA и SPA в единое целое. (Такое использование термина отчасти напоминает «переходный» стиль в архитектуре, сочетающий традиционные и современные архитектурные стили.)

Термин «переходный» подходит для приложений смешанного стиля. Он обеспечивает разумный компромисс между двумя подходами, позволяя использовать наиболее подходящий вариант в каждом конкретном случае.

Тем не менее такой компромисс все еще остается неудовлетворительным.

Нужно ли включать эти две очень разные архитектурные модели в наши приложения по умолчанию?

Вспомните, что принципиальным моментом выбора между SPA и MPA является *опыт взаимодействия с пользователем*, или интерактивность приложения. Обычно этот фактор заставляет выбирать тот или иной подход для приложения или (в случае «переходного» приложения) для отдельной функциональности.

Оказывается, выбор гипермедиа-ориентированной библиотеки радикально сокращает разрыв в интерактивности между подходами MPA и SPA. Вы можете использовать подход MPA (то есть гипермедийный подход) для гораздо большей части приложения без ущерба для пользовательского интерфейса. Возможно, вам даже удастся использовать гипермедийный подход для *всех* потребностей приложения.

Вместо того чтобы создавать приложение SPA с гипермедийными довесками или комбинацию двух подходов, часто можно написать веб-приложение, которое *в основном* или *полностью* является гипермедиа-управляемым и которое предоставляет всю интерактивность, необходимую пользователям.

Это позволяет добиться *невероятной* простоты веб-приложения и создать намного более целостный и понятный продукт. Хотя в каких-то ситуациях все еще может применяться более сложный подход SPA, что мы обсудим позже, переход к решению, ставящему на первое место гипермедийность, и применение гипермедиа-ориентированной библиотеки для расширения возможностей HTML позволяют строить мощные, интерактивные *и* простые веб-приложения.

К числу таких гипермедиа-ориентированных библиотек относится `htmx`[1]. Библиотеке `htmx` будет посвящена часть II этой книги. Мы покажем, что многие популярные «современные» средства UI, присутствующие в сложных приложениях SPA, можно реализовать на основе модели гипермедиа.

И делать это на удивление просто и интересно.

[1] https://htmx.org/

Гипермедиа-управляемые приложения

При построении веб-приложения с `htmx` термин «многостраничное приложение» применим *в общих чертах*, но он не характеризует основу архитектуры приложения в полной мере. Как вы вскоре увидите, *не обязательно* заменять `htmx` целые страницы; более того, приложение на основе `htmx` может существовать полностью в пределах одной страницы. Мы не рекомендуем такую практику, но это возможно!

Таким образом, называть веб-приложения, построенные на основе `htmx`, «многостраничными» не совсем корректно. У старого подхода MPA для приложений Web 1.0 и у приложений на основе новых гипермедиа-ориентированных библиотек есть нечто общее: они используют *гипермедиа* как базовую технологию и архитектуру.

По этой причине мы будем использовать термин *гипермедиа-управляемые приложения*, или *HDA* (Hypermedia-Driven Applications), для описания обеих разновидностей.

Это определение проясняет основное различие между этими двумя подходами, а подход SPA определяет *не* количество страниц в приложении, а скорее используемую архитектуру *системы*.

Гипермедиа-управляемое приложение (HDA)

> Веб-приложение, использующее *гипермедиа* и *гипермедийный обмен данными* в качестве основного механизма взаимодействия с сервером.

Как же выглядит приложение HDA при ближайшем рассмотрении?

Перед вами реализация простой JavaScript-кнопки с использованием `htmx`.

Листинг 5. Реализация htmx

```
<button hx-get="/contacts/1" hx-target="#contact-ui"> ❶
  Fetch Contact
</button>
```

❶ Выдает запрос GET к /contacts/1, заменяя contact-ui.

Как и у кнопки на основе JavaScript, у этой кнопки существуют атрибуты. Однако в этом случае не используется (во всяком случае, явно) скриптовый код JavaScript.

Вместо него используются *декларативные* атрибуты, очень похожие на атрибуты `href` якорных тегов, или атрибуты `action` тегов форм. Атрибут `hx-get` сообщает `htmx`: «Когда пользователь щелкает на этой кнопке, отправить запрос `GET` к `/contacts/1`». Атрибут `hx-target` сообщает `htmx`: «Когда будет возвращен

ответ, принять полученную разметку HTML и поместить ее в элемент с идентификатором `contact-ui`».

Здесь мы подходим к сути `htmx` и тому, как с помощью этой библиотеки строить гипермедиа-управляемые приложения.

Ожидается, что сервер вернет ответ HTTP в формате HTML, а не JSON.

Ответ HTTP на этот запрос под управлением `htmx` может выглядеть примерно так:

Листинг 6. JSON

```
<details>
 <div>
  Contact: HTML Example
 </div>
 <div>
  <a href="mailto:html-example@example.com">Email</a>
 </div>
</details>
```

Этот маленький фрагмент HTML будет помещен в элемент DOM с идентификатором `contact-ui`.

Таким образом, кнопка на основе `htmx` обменивается с сервером данными *гипермедиа* (как мог бы делать якорный тег или тег формы), и следовательно, взаимодействие ведется в соответствии с базовой гипермедиа-моделью веб-среды. `Htmx` добавляет к кнопке функциональность (при помощи JavaScript), но эта функциональность *дополняет* HTML как язык гипермедиа. `Htmx` расширяет гипермедиа-систему веб-среды, а не *заменяет* эту систему совершенно иной архитектурой.

Хотя внешне две реализации очень похожи, кнопки на основе `htmx` и на основе JavaScript используют совершенно разные системные архитектуры и, как следствие, разные подходы к веб-разработке.

По мере того как мы будем разбирать построение приложений HDA, различия между двумя подходами будут становиться все более явными.

Когда использовать гипермедиа?

Гипермедиа часто, хотя и не всегда, отлично подходит для создания веб-приложения.

Возможно, вы строите сайт или приложение, которым просто не нужен высокий уровень интерактивности. Полезных веб-приложений такого рода очень много, и в этом нет ничего плохого! Таким приложениям, как Amazon, eBay, многим

новостным сайтам, интернет-магазинам, форумам и т. д., просто не нужна избыточная интерактивность: они в основном состоят из текста и графики, а ведь это именно то, для чего создавался веб.

Возможно, бóльшая часть ценности приложения создается *на стороне сервера* (координация пользователей, сложный анализ данных) с последующим представлением результатов пользователю. А может быть, приложение создает ценность, используя хорошо спроектированную базу данных с простыми операциями CRUD (Create-Read-Update-Delete, создать-прочесть-обновить-удалить). И здесь опять-таки нечего стыдиться!

Во всех этих случаях гипермедиа-решение может стать отличным вариантом: потребности таких приложений в интерактивности невелики, а большая часть их ценности сосредоточена на стороне сервера, а не на стороне клиента.

Во всех описанных случаях применимо то, что Рой Филдинг называл «крупномодульной гипермедийной передачей данных»: можно просто использовать якорные теги и теги форм с ответами, возвращающими целые документы HTML по запросам, и все будет работать нормально. Именно для этого создавался веб.

Применяя гипермедиа в таких приложениях, вы избавите себя от значительной сложности на стороне клиента, которая неизбежно сопровождает решения SPA: нет необходимости в маршрутизации на стороне клиента, управлении моделью на стороне клиента, ручном подключении логики JavaScript и т. д. Кнопка возврата «просто работает». Глубокие ссылки «просто работают». Вы сможете сосредоточить свои усилия на сервере, где создается реальная ценность приложения.

Кроме того, наложение `htmx` или другой гипермедиа-ориентированной библиотеки поверх такого решения позволит избавиться от многих проблем с удобством использования, присущих базовому HTML, и осуществлять более детализированную передачу данных гипермедиа. Все это открывает новые возможности для создания пользовательского интерфейса и взаимодействий, что *значительно* расширяет набор приложений, которые можно создавать на основе гипермедиа.

Впрочем, подробности позже.

Когда не стоит использовать гипермедиа?

Когда же гипермедиа не будет работать?

Пример, который сразу приходит на ум, — электронная онлайн-таблица. В электронной таблице обновление одной ячейки может привести ко множеству каскадных изменений во всей таблице. Что еще хуже, это может происходить *при каждом нажатии клавиши*.

В этом случае мы имеем чрезвычайно динамичный пользовательский интерфейс без четких ограничений по обновлениям, вызываемым конкретным изменением.

Круговой обмен данных с сервером, характерный для гипермедиа, приведет к громадной потере производительности.

Эта ситуация не подходит для «крупномодульной гипермедийной передачи данных» в веб-среде. В таких приложениях, безусловно, стоит рассмотреть решение с нестандартным кодом JavaScript на стороне клиента.

Тем не менее даже в приложении электронной онлайн-таблицы, вероятно, найдутся области, в которых гипермедиа может принести пользу.

Скорее всего, в электронной таблице есть страница настроек. И возможно, к ней можно применить гипермедийный подход. Если эта страница представляет собой набор относительно простых форм, которые должны сохраняться на сервере, гипермедиа, очевидно, отлично подойдет для этой части приложения.

Кроме того, переход на гипермедиа позволит значительно упростить эту часть приложения. Возможно, удастся направить большую часть *бюджета сложности* приложения на нетривиальную рабочую логику электронной таблицы, сохраняя простые компоненты простыми.

Зачем применять сложный, мощный фреймворк JavaScript к чему-то столь же элементарному, как страница настроек?

БЮДЖЕТ СЛОЖНОСТИ

У любого программного проекта имеется (явно или неявно) бюджет сложности: объем сложности, с которым справляется команда разработки, ограничен, и каждая новая функциональность или выбор в процессе реализации хоть немного, но повышают общую сложность системы.

Главный минус сложности в том, что она стремится к экспоненциальному росту: сегодня вы держите всю систему у себя в голове и понимаете последствия конкретного изменения, а через неделю система выходит из-под контроля. Что еще хуже, усилия по контролю над сложностью (например, введение абстракций или инфраструктуры для управления сложностью) часто только повышают ее. Задача хорошего разработчика – удержать сложность под контролем.

Самый надежный способ ограничить сложность также и самый трудный: научиться говорить «нет». Отклонять запросы на добавление функциональности – настоящее искусство. Если вы научитесь делать это так, чтобы людям казалось, будто *они сами* сказали «нет», вы далеко пойдете.

Как ни печально, это получается не всегда: некоторая функциональность просто необходима. Тогда поставьте вопрос так: «Каково самое простое решение, которое будет работать?» Понимание возможностей гипермедиа-подхода дополнит ваш инструментарий «простейших решений» новыми средствами.

Гипермедиа: комплексная современная системная архитектура

В веб-разработке гипермедиа часто считают устаревшей, изжившей себя технологией — возможно, полезной для статических сайтов, но определенно не подходящей для современных комплексных веб-приложений.

Серьезно? Мы заявляем, что на основе этой технологии можно строить современные веб-приложения?

Да, серьезно.

В отличие от расхожего мнения, гипермедиа — это *инновационная* и *современная* системная архитектура для построения приложений, в некоторых отношениях *более современная*, чем популярные решения SPA. В оставшейся части этой книги мы заново познакомим вас с основными практическими концепциями гипермедиа, а затем покажем, как пользоваться этой системной архитектурой для создания продуктов.

В следующих главах мы постараемся объяснить все преимущества и приемы, используемые в этом подходе. Надеемся, вы станете таким же поклонником гипермедиа, как и мы.

Заметки об HTML: каша из

Самый известный пример грязного HTML — каша из `<div>`.

Когда разработчикам приходится пользоваться обобщенными элементами `<div>` и `<span>` вместо более содержательных тегов, это либо приводит к снижению качества их веб-сайтов, либо они делают лишнюю работу, а скорее всего, и то и другое.

Например, вместо добавления кнопки в виде специализированного элемента `<button>` к элементу `<div>` может быть присоединен прослушиватель события `click`.

```
<div class="bg-accent padding-4 rounded-2" onclick="doStuff()">Сделать
что-нибудь</div>
```

У такой кнопки два основных недостатка.

- Она не может получить фокус — к ней нельзя перейти клавишей Tab.
- Вспомогательные инструменты разработки не смогут определить, что это кнопка.

Да, проблемы можно решить добавлением атрибутов `role="button"` и `tabindex="0"`:

```
<div class="bg-accent padding-4 rounded-2"
 role="button"
 tabindex="0"
 onclick="doStuff()">Сделать что-нибудь</div>
```

Это простые решения, но их приходится *помнить*. Из исходного кода HTML не очевидно, что это кнопка. Поэтому будет сложнее прочитать код и заметить отсутствие этих атрибутов. Исходный код страниц с кашей из `<div>` труднее редактировать и отлаживать.

Чтобы избежать создания каши из `<div>`, мы рекомендуем изучить спецификацию доступных тегов HTML и относиться к каждому тегу как к еще одному инструменту из доступного набора. Вы даже можете узнать что-то новое для себя! (Хотя если учесть, что на сегодняшний день в спецификации определено 113 элементов, это скорее мастерская, а не ящик с инструментами.)

Конечно, не для каждого UI-паттерна найдется специальный элемент HTML. Часто приходится объединять элементы и дополнять их атрибутами. Но прежде чем делать это, перетряхните свой инструментарий HTML. Просто невероятно, сколько полезного в нем можно найти.

КОМПОНЕНТЫ СИСТЕМЫ ГИПЕРМЕДИА

Система гипермедиа состоит из нескольких компонентов, в числе которых:

- среда гипермедиа, например HTML;
- сетевой протокол, например HTTP;
- сервер, предоставляющий API гипермедиа, посылающий гипермедиа в ответ на сетевые запросы;
- клиент, корректно интерпретирующий эти ответы.

В этой главе мы рассмотрим эти компоненты и их реализацию в контексте веб-среды.

После обзора основных компонентов веб-среды как системы гипермедиа мы перейдем к рассмотрению некоторых идей, лежащих в основе этой системы, — в том виде, в каком они были разработаны Роем Филдингом в его диссертации «Architectural Styles and the Design of Network-based Software Architectures». Вы узнаете, откуда взялись термины REST (REpresentational State Transfer), RESTful и HATEOAS (Hypermedia As The Engine Of Application State). Мы проанализируем эти термины в контексте веб-среды.

Все это поможет вам глубже понять теоретическую основу Всемирной паутины как системы гипермедиа, предпочтительную схему взаимодействия ее компонентов и то, почему гипермедиа-управляемые приложения являются RESTful-приложениями, а JSON API — нет (невзирая на то, как сейчас в отрасли принято использовать термин REST).

Среда гипермедиа

Фундаментальной технологией системы гипермедиа является среда гипермедиа, которая позволяет клиенту и серверу динамически и нелинейно взаимодействовать друг с другом. Напомним, что гипермедиа делает таковым именно присутствие *элементов управления гипермедиа*, то есть элементов, позволяющих пользователям выбирать нелинейные действия в гипермедиа-среде. Возмож-

ности *взаимодействия* пользователя с информационной средой не ограничиваются простым чтением от начала к концу.

Мы уже упоминали о двух основных гипермедийных элементах управления HTML — якорях и формах, позволяющих представлять ссылки и операции пользователю непосредственно через браузер.

В случае HTML эти ссылки и формы обычно задают цель своих операций в форме *унифицированных указателей ресурсов*, или *URL* (Uniform Resource Locators):

> *Унифицированный указатель ресурсов* — текстовая строка, ссылающаяся, или *указывающая*, на сетевой каталог, из которого может быть загружен ресурс, а также механизм загрузки этого ресурса.

URL представляет собой строку, состоящую из нескольких подкомпонентов.

Листинг 7. Компоненты URL

```
[схема]://[информация_пользователя]@[хост]:[порт][путь]?[запрос]#[фрагмент]
```

Многие из этих подкомпонентов необязательны и часто опускаются.

Типичный URL может выглядеть примерно так:

Листинг 8. Простой URL

```
https://hypermedia.systems/book/contents/
```

Этот конкретный URL состоит из следующих компонентов:

- Протокол или схема (в данном случае `https`);
- домен (в данном случае `hypermedia.systems`);
- путь (в данном случае `/book/contents`).

Этот URL однозначно идентифицирует *ресурс*, доступный в интернете, запросы HTTP к которому могут отправляться клиентом гипермедиа, поддерживающим HTTPS, например веб-браузером. Если этот URL представлен как ссылка гипермедийного элемента управления внутри документа HTML, это указывает на то, что на другом конце сети находится *сервер гипермедиа*, который также поддерживает HTTPS и может ответить на запрос *представлением* указанного ресурса (или перенаправить в другое место и т. д.).

Учтите, что URL часто не записываются в HTML полностью. Очень часто встречаются якорные теги, которые могут выглядеть примерно так:

Листинг 9. Простая ссылка

```
<a href="/book/contents/">Содержание</a>
```

Здесь используется *относительная* гипермедийная ссылка, которая, *как предполагается*, содержит протокол, хост и порт «текущего документа», то есть протокол и сервер, использованные для загрузки текущей страницы HTML. Таким образом, если эта ссылка будет найдена в документе HTML, загруженном по адресу https://hypermedia.systems/, то предполагаемый URL для этого якорного тега будет иметь вид https://hypermedia.systems/book/contents/.

Гипермедиа-протоколы

Приведенный выше элемент гипермедиа (ссылка) сообщает браузеру: «Когда пользователь щелкает на этом тексте, отправить запрос к https://hypermedia.systems/book/contents/ с помощью протокола HTTP».

HTTP — *протокол*, используемый для передачи HTML (гипермедиа) между браузерами (клиентами гипермедиа) и серверами (серверами гипермедиа); как следствие, он также является ключевой сетевой технологией, связывающей воедино распределенные гипермедиа-системы веб-среды.

HTTP версии 1.1 — относительно простой сетевой протокол, поэтому посмотрим, как будет выглядеть запрос `GET`, инициированный якорным тегом. Этот запрос будет отправляться серверу, найденному по адресу `hypermedia.systems`, на порт 80 по умолчанию:

```
GET /book/contents/ HTTP/1.1
Accept: text/html,*/*
Host: hypermedia.systems
```

Первая строка сообщает, что это запрос HTTP `GET`, и указывает путь к запрашиваемому ресурсу. Строка завершается версией HTTP для запроса.

Далее следует серия *заголовков запроса* HTTP: отдельные строки с парами «имя/значение», разделенными двоеточием. Заголовки запроса предоставляют метаданные, которые могут использоваться сервером для определения того, какой ответ возвращать на запрос клиента. В данном случае при помощи заголовка `Accept` браузер указывает предпочтительный формат ответа — HTML, но также сообщает, что примет любой ответ сервера.

Затем идет заголовок `Host`, который сообщает, какому серверу был отправлен запрос. Эта информация может быть полезной, если на одном хосте размещаются несколько доменов.

Ответ HTTP от сервера на запрос может выглядеть примерно так:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 870
```

```
Server: Werkzeug/2.0.2 Python/3.8.10
Date: Sat, 23 Apr 2022 18:27:55 GMT

<html lang="en">
<body>
 <header>
  <h1>СИСТЕМЫ ГИПЕРМЕДИА</h1>
 </header>
 ...
</body>
</html>
```

В первой строке ответа HTTP указывается используемая версия HTTP, за которой следует *код ответа* 200; он означает, что указанный ресурс был найден, а запрос обработан успешно. Далее следует строка `OK`, которая соответствует коду ответа. (Фактическая строка роли не играет, результат запроса сообщается клиенту в виде кода ответа — более подробно мы разберем это ниже.)

После первой строки ответа, как и в запросе HTTP, следует серия *заголовков ответа*, которые предоставляют клиенту метаданные, помогающие обеспечить корректное *представление* ресурса.

В конце мы видим новый контент HTML. Контент содержит *представление* запрашиваемого ресурса в формате HTML, в данном случае оглавление книги. Браузер использует эту разметку HTML для замены всего содержимого отображаемого окна, показывает пользователю новую страницу и обновляет адресную строку новым URL.

Методы HTTP

Приведенный выше якорный тег выдает запрос HTTP `GET`, где `GET` — *метод* запроса. Метод, указанный в запросе HTTP, по сути, содержит самую важную информацию о запросе (после собственно ресурса, к которому обращен запрос).

В HTTP доступно множество методов; чаще всего разработчикам могут пригодиться следующие из них:

- Запрос `GET` получает представление конкретного ресурса. Запросы `GET` не должны изменять данные.
- Запрос `POST` отправляет данные указанному ресурсу. Часто это приводит к изменению состояния на сервере.
- Запрос `PUT` заменяет данные указанного ресурса. Это приводит к изменению состояния на сервере.

- Запрос `PATCH` заменяет данные указанного ресурса. Это приводит к изменению состояния на сервере.
- Запрос `DELETE` удаляет указанный ресурс. Это приводит к изменению состояния на сервере.

Эти методы схожи с операциями схемы CRUD (Create/Read/Update/Delete), встречающейся во многих приложениях:

- `POST` соответствует созданию (Create) ресурса;
- `GET` соответствует чтению (Read) ресурса;
- `PUT` и `PATCH` соответствуют обновлению (Update) ресурса;
- `DELETE` соответствует (очевидно) удалению (Delete) ресурса.

PUT И POST

Хотя методы HTTP схожи с операциями CRUD, они не эквивалентны. В технических спецификациях методов такие связи не прослеживаются, из-за чего их довольно трудно читать. Например, ниже приведен фрагмент документации о различиях между POST и PUT из RFC-2616[1].

> “Целевой ресурс в запросе POST должен обрабатывать включенное представление в соответствии с собственной семантикой ресурса, тогда как включенное представление в запросе PUT определяется как замена состояния целевого ресурса. Как следствие, назначение PUT идемпотентно и видимо посредникам, хотя точный результат известен только серверу-источнику.
>
> ~ *RFC-2616* https://www.rfc-editor.org/rfc/rfc2616#section-9.6

Проще говоря, сервер может обрабатывать запрос POST практически произвольно, тогда как запрос PUT должен обрабатываться как «замена» ресурса, хотя формулировка также наделяет сервер свободой действий, в пределах ограничений идемпотентности[2].

В грамотно структурированной системе гипермедиа на основе HTML для операции, выполняемой конкретным элементом управления, должен использовать-

[1] https://www.rfc-editor.org/rfc/rfc2616

[2] https://developer.mozilla.org/en-US/docs/Glossary/Idempotent

ся определенный метод HTTP. Например, если элемент управления (скажем, кнопка) *удаляет* ресурс, в идеале он должен выдавать для этого запрос HTTP `DELETE`.

Однако у HTML есть одна странность: нативные элементы управления могут выдавать только запросы HTTP `GET` и `POST`.

Якорные теги всегда выдают запрос `GET`.

Формы могут выдавать либо запрос `GET`, либо запрос `POST` в зависимости от атрибута `method`.

Несмотря на то что HTML, самая популярная среда гипермедиа в мире, проектировался с учетом специфики HTTP (ведь это протокол передачи гипертекста, в конце концов!), если вам необходимо выдать запрос `PUT`, `PATCH` или `DELETE`, *придется* воспользоваться JavaScript. Так как запрос `POST` может делать практически все что угодно, в конечном итоге он применяется для любых изменений на сервере, а запросы `PUT`, `PATCH` и `DELETE` не используются в простых приложениях на основе HTML.

Это очевидный недостаток HTML как гипермедиа; было бы здорово исправить его в спецификации HTML. А пока в главе 4 будут рассмотрены некоторые обходные решения этой проблемы.

Коды ответа HTTP

Методы запросов HTTP позволяют клиенту сообщить серверу, *что* делать с указанным ресурсом. Ответы HTTP содержат *коды ответов*, по которым клиент определяет результат запроса. Коды ответов HTTP представляют собой числовые значения, встроенные в ответ HTTP, как было показано выше.

Веб-разработчики наверняка лучше всего знакомы с кодом ответа 404, означающим, что ресурс не найден. Этот код ответа веб-серверы возвращают, когда у них запрашивается несуществующий ресурс.

В HTTP коды ответов делятся на категории:

- `100-199` — содержат информацию о том, как сервер обрабатывает ответ;
- `200-299` — сообщают об успешной обработке запроса;
- `300-399` — коды перенаправления, указывающие, что запрос должен быть отправлен по другому URL;
- `400-499` — коды ошибок клиента, указывающие, что клиент выдал некорректный запрос (например, запросил несуществующий ресурс в случае ошибки 404);

- `500-599` — коды ошибок сервера, указывающие, что при попытке ответить на запрос на сервере произошла внутренняя ошибка.

Каждая категория содержит коды ответов для конкретных ситуаций.

Несколько самых распространенных или интересных кодов ответов:

- `200 OK` — запрос HTTP обработан успешно;
- `301 Moved Permanently (Перемещено навсегда)` — запрашиваемый ресурс навсегда перемещен в новое место, и новый URL предоставляется в заголовке ответа `Location`;
- `302 Found (Найдено)` — запрашиваемый ресурс временно перемещен в новое место, и новый URL предоставляется в заголовке ответа `Location`;
- `303 See Other (Смотреть другое)` — URL запрашиваемого ресурса перемещен в новое место, и новый URL предоставляется в заголовке ответа `Location`. Кроме того, для получения нового URL следует использовать запрос `GET`;
- `401 Unauthorized (Не авторизован)` — клиент еще не прошел аутентификацию (да, аутентификацию — несмотря на название), необходимую для получения заданного ресурса;
- `403 Forbidden (Запрещено)` — у клиента нет доступа к запрашиваемому ресурсу;
- `404 Not Found (Не найдено)` — сервер не может найти запрашиваемый ресурс;
- `500 Internal Server Error (Внутренняя ошибка сервера)` — на сервере произошла ошибка при попытке обработать запрос.

Между кодами ответов HTTP существуют довольно тонкие различия (и, откровенно говоря, неоднозначности). Например, различие между перенаправлениями `302` и `303` заключается в том, что первое выдает запрос к новому URL с использованием такого же метода HTTP, как в исходном запросе, тогда как второе всегда использует `GET`. Это небольшое различие часто оказывается критичным, как вы увидите далее.

Хорошо построенное гипермедиа-управляемое приложение использует как методы HTTP, так и коды ответа HTTP для создания грамотного API гипермедиа. Например, вам вряд ли нужно гипермедиа-управляемое приложение, использующее метод `POST` для всех запросов и отвечающее кодом `200 OK` на каждый запрос. (Хотя некоторые JSON Data API, построенные на основе HTTP, именно так и делают!)

При создании гипермедиа-управляемых приложений лучше пойти «по течению» веб-технологий и использовать методы HTTP и коды ответов способом, который был заложен при их проектировании.

Кэширование ответов HTTP

Одним из ограничений REST (а следовательно, и рабочих механизмов HTTP) является концепция кэширования запросов: сервер сообщает клиенту (а также промежуточным серверам HTTP), что ответ может быть кэширован для будущих запросов к тому же URL.

Для управления поведением кэширования ответов HTTP от сервера используется заголовок ответа `Cache-Control`. Этот заголовок может принимать разные значения, описывающие возможность кэширования данного ответа. Например, если заголовок содержит значение `max-age=60`, это означает, что клиент может кэшировать этот ответ в течение 60 секунд и ему не придется выдавать другой запрос HTTP к этому ресурсу до истечения этого времени.

Еще один важный заголовок ответа, относящийся к кэшированию, — `Vary`. Он используется для обозначения того, какие заголовки запроса HTTP формируют уникальный идентификатор кэшированного результата. Это важно, чтобы браузер мог правильно кэшировать контент в ситуациях, когда конкретный заголовок влияет на формирование ответа сервера.

Например, в приложениях на основе `htmx` часто встречается такой паттерн: использование нестандартного заголовка `HX-Request`, назначаемого `htmx`, для того чтобы отличать «обычные» веб-запросы от запросов, отправленных `htmx`. Чтобы правильно кэшировать ответ на эти запросы, заголовок запроса `HX-Request` должен быть обозначен в заголовке ответа `Vary`.

Полное обсуждение кэширования ответов HTTP выходит за рамки этой главы; чтобы узнать больше по этой теме, обратитесь к статье MDN о кэшировании HTTP[1].

Серверы гипермедиа

Сервером гипермедиа является любой сервер, способный на запрос HTTP отправить ответ HTTP. Так как протокол HTTP очень прост, это означает, что для создания сервера гипермедиа можно использовать практически любой язык программирования. Почти для каждого известного языка существует множество библиотек для построения серверов гипермедиа на основе HTTP.

И это один из лучших аргументов в пользу гипермедиа как основной технологии построения веб-приложений: это избавляет от необходимости применять JavaScript для бэкенда. Если фронтенд вашего одностраничного приложения основан на JavaScript и, помимо этого, вы используете JSON Data API, вы будете вынуждены использовать JavaScript и в бэкенде.

---

[1] https://developer.mozilla.org/en-US/docs/Web/HTTP/Caching

В этом случае у вас уже есть масса кода, написанного на JavaScript. Зачем поддерживать две отдельные кодовые базы на двух разных языках? Почему не создать логику предметной области, пригодную для повторного использования как на стороне клиента, так и на стороне сервера? Для JavaScript доступны отличные технологии на стороне сервера (такие, как Node и Deno), так почему бы не использовать один язык для всего?

С другой стороны, гипермедиа-управляемые приложения предоставляют намного больше свободы в выборе технологии бэкенда. Ваше решение может зависеть от предметной области приложения, известных вам языков и серверных продуктов (или ваших личных предпочтений) или вообще от того, что вы хотите попробовать.

Разумеется, вы не станете писать логику сервера на HTML! И в каждом крупном языке программирования существуют хотя бы один хороший веб-фреймворк и библиотека шаблонов, которые можно использовать для качественной обработки запросов HTTP.

Если вы работаете с большими данными, возможно, вам стоит выбрать Python с его отличной поддержкой в области data science.

Если вы работаете над проектом в сфере искусственного интеллекта, возможно, вы предпочтете Lisp — язык с долгой историей в этой области исследований.

А может, вы энтузиаст функционального программирования и хотите использовать OCaml или Haskell. Или вам больше нравится Julia или Nim.

Все это абсолютно обоснованные причины для выбора конкретной серверной технологии!

СТЕК HOWL

В сообществе `htmx` мы называем такой подход «стеком HOWL» (Hypermedia On Whatever you'd Like, то есть «гипермедиа на чем угодно»). В сообществе htmx используются разные языки и фреймворки, в нем есть любители как Ruby, так и Python, Lisp и Haskell. Найдутся даже энтузиасты JavaScript! Все эти языки и фреймворки могут перейти на гипермедиа, сохраняя возможность обмениваться полезной информацией и предоставлять поддержку друг другу, потому что они основаны на единой нижележащей архитектуре: все они используют Паутину как систему гипермедиа.

В этом смысле гипермедиа предоставляет «универсальный язык» для веб-среды, который могут использовать все.

Выбирая гипермедиа в качестве архитектуры системы, вы избавляетесь от необходимости подстраиваться под любой из этих вариантов. На фронтенде по-

просту не будет большого объема кода JavaScript, который вынудил бы вас адаптировать JavaScript для бэкенда.

Клиенты гипермедиа

Мы подходим к последнему основному компоненту гипермедиа-системы: клиенту гипермедиа. *Клиенты* гипермедиа представляют собой программы, которые способны правильно интерпретировать конкретную среду гипермедиа и содержащиеся в ней элементы управления. Классическим примером служит веб-браузер, который понимает HTML и отображает его для взаимодействия с пользователем. Веб-браузеры невероятно сложны (настолько, что часто из клиентов гипермедиа они вырастают в своего рода кроссплатформенные виртуальные машины для запуска одностраничных приложений).

Впрочем, браузеры — не единственные существующие клиенты гипермедиа. В заключительной части книги рассматривается Hyperview, мобильно-ориентированная гипермедийная платформа. Одна из ее замечательных особенностей заключается в том, что Hyperview предоставляет не просто гипермедиа, HXML, но и *работоспособный клиент гипермедиа* для этой среды. Это невероятно упрощает создание гипермедиа-управляемых приложений.

Важнейшей особенностью системы гипермедиа становится так называемый *унифицированный интерфейс*. Это понятие более подробно рассматривается в следующем разделе, посвященном REST. При обсуждении гипермедиа часто упускают из виду, насколько важную роль играет клиент гипермедиа в использовании унифицированного интерфейса. Чтобы все компоненты системы гипермедиа нормально взаимодействовали друг с другом, клиент гипермедиа должен уметь правильно интерпретировать и представлять элементы управления, содержащиеся в ответе сервера гипермедиа. Без тщательно проработанного клиента, который умеет это делать, ценность элементов управления и API на основе гипермедиа будет снижена.

Это одна из причин, почему JSON API редко успешно используют элементы гипермедиа: JSON API обычно потребляются кодом, рассчитанным на фиксированный формат, а не на выполнение функций клиента гипермедиа. И это вполне понятно: построить хороший клиент гипермедиа трудно! Для таких клиентов JSON API возможности элементов гипермедиа, встроенных в ответ API, неактуальны и часто даже раздражают:

> “Короткий ответ на этот вопрос: HATEOAS попросту недостаточно хорошо подходит для современных сценариев использования API. Вот почему по прошествии почти 20 лет HATEOAS так и не получил

> широкого распространения среди разработчиков. С другой стороны, GraphQL распространяется со скоростью лесного пожара, поскольку решает реальные задачи.
>
> *Фредди Карлбом (Freddie Karlbom),*
> https://techblog.commercetools.com/graphql-and-rest-level-3-hateoas-70904ff1f9cf

Концепция HATEOAS более подробно рассматривается ниже, но здесь важно то, что хороший клиент гипермедиа является необходимым компонентом крупной гипермедиа-системы.

REST

После обзора всех основных компонентов системы гипермедиа пришло время более подробно рассмотреть концепцию REST. Термин «REST» был введен в диссертации Роя Филдинга, посвященной веб-архитектуре. Филдинг писал свою диссертацию в Калифорнийском университете в Ирвайне после того, как внес значительный вклад в создание ранней веб-инфраструктуры, включая разработку веб-сервера Apache. Рой попытался формализовать и описать новаторскую распределенную вычислительную систему, которую он помогал строить.

Мы сосредоточимся на том, что нам кажется самой важной частью работы Филдинга с точки зрения веб-разработки: разделе 5.1. В нем содержатся основные концепции (Филдинг называл их *«ограничениями»*) передачи состояния представления, или *REST*.

Но прежде чем переходить к подробностям, важно понять, что Филдинг говорит о REST как о *сетевой архитектуре*, то есть совершенно ином способе проектирования распределенной системы. И кроме того, об инновационной сетевой архитектуре, которая *противопоставляется* более ранним подходам к построению распределенных систем.

Важно заметить, что в то время, когда Филдинг писал свою диссертацию, JSON API и AJAX еще не существовали. Он описывал раннюю веб-среду, в которой разметка HTML передавалась по протоколу HTTP первыми браузерами, как систему гипермедиа.

Но события развернулись так, что термин REST теперь в основном ассоциируется с JSON Data API, а не с HTML и гипермедиа. Это очень забавно, если осознать, что в подавляющем большинстве JSON Data API не соответствуют принципам REST в исходном смысле; и более того, не могут им соответствовать, так как изначально не используют формат гипермедиа.

Подчеркнем еще раз: термин REST в том смысле, в каком его вводит Филдинг, описывает *веб-среду до появления API*, и для правильного понимания этой кон-

цепции необходимо отказаться от современного, широко распространенного использования термина REST в значении JSON API.

«Ограничения» REST

В своей диссертации Филдинг определяет различные «ограничения» для описания поведения, которым должны обладать RESTful-системы. Многим этот подход может показаться излишне замысловатым и трудным для понимания, но это уместно для академического труда. Если немного поразмыслить над ограничениями, которые он описывает, и конкретными примерами таких ограничений, легко оценить, удовлетворяет заданная система архитектурным требованиям REST или нет.

Ограничения REST, описанные Филдингом:

- Это архитектура «клиент — сервер» (раздел 5.1.2).
- Она не должна иметь состояния (раздел 5.1.3), то есть каждый запрос должен содержать всю информацию, необходимую для ответа на этот запрос.
- Она должна допускать кэширование (раздел 5.1.4).
- Она должна иметь *унифицированный интерфейс* (раздел 5.1.5).
- Это многоуровневая система (раздел 5.1.6).
- Она может дополнительно поддерживать возможность отправки кода по запросу (Code-On-Demand, раздел 5.1.7), то есть передачу скриптового кода.

Рассмотрим все эти ограничения поочередно и обсудим их более подробно, уделяя особое внимание тому, как (и в какой степени) веб-среда удовлетворяет каждое из них.

Ограничение «клиент-сервер»

Ограничение рассмотрено в разделе 5.1.2[1].

В модели REST, описанной Филдингом, участвуют как *клиенты* (браузеры в случае веб-среды), так и *серверы* (например, веб-сервер Apache, над которым он работал), взаимодействующие через сетевое соединение. Это было контекстом работы Филдинга: он описывал сетевую архитектуру Всемирной паутины и сравнивал ее с более ранними архитектурами, прежде всего сетевыми моделями толстого клиента, такими как CORBA (Common Object Request Broker Architecture).

Очевидно, что любое веб-приложение, независимо от того, как оно спроектировано, удовлетворяет этому требованию.

---

[1] https://ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_1_2

Ограничение отсутствия состояния

Ограничение рассмотрено в разделе 5.1.3[1].

Согласно описанию Филдинга, RESTful-система не обладает состоянием: каждый запрос должен инкапсулировать всю информацию, необходимую для ответа на него, без побочного состояния или контекста, хранимого на стороне клиента или сервера.

На практике для многих веб-приложений это ограничение нарушается: часто создается *сеансовый объект cookie*, который действует как уникальный идентификатор для конкретного пользователя, и пересылается с каждым запросом. Хотя cookie сам по себе не обладает состоянием (он отправляется с каждым запросом), как правило он используется в качестве ключа для обращения к информации, хранимой на сервере, в том, что обычно называется «сеансом».

Информация сеанса обычно хранится в разновидности общего хранилища для разных веб-серверов, в котором содержатся такие данные, как электронная почта или идентификатор текущего пользователя, его роли, частично созданные объекты предметной области, кэшированные данные и т. д.

Это нарушение архитектурного ограничения отсутствия состояния REST оказалось полезным для построения веб-приложений, и похоже, оно не оказывает значительного влияния на общую гибкость веб-среды. Однако следует учитывать, что даже приложения Web 1.0 часто нарушают теоретическую чистоту REST в практических интересах.

Также необходимо сказать, что сеансы создают дополнительную эксплуатационную сложность при развертывании серверов гипермедиа; им может понадобиться общий доступ к информации состояния сеанса, хранимой в пределах целого кластера. Таким образом, Филдинг совершенно справедливо указывал, что идеальная RESTful-система, не нарушающая это ограничение, будет более простой и, следовательно, более устойчивой.

Ограничение кэширования

Ограничение рассмотрено в разделе 5.1.4[2].

Это ограничение утверждает, что RESTful-система должна поддерживать механизм кэширования, с явной информацией о пригодности ответов к кэшированию для будущих запросов к тому же ресурсу. Это позволяет как клиентам, так и промежуточным серверам между заданным клиентом и итоговым сервером кэшировать результаты запросов.

---

[1] https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_1_3

[2] https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_1_4

Как говорилось выше, в HTTP поддерживается комплексный механизм кэширования с использованием заголовков ответов, о котором часто забывают или не полностью задействуют при построении приложений гипермедиа. Впрочем, так как эта функциональность существует, убедиться в соблюдении такого ограничения веб-средой легко.

Ограничение унифицированного интерфейса

Мы подошли к самому интересному и, на наш взгляд, самому инновационному ограничению в REST: ограничению *унифицированного интерфейса*.

Это ограничение обеспечивает значительную часть гибкости и простоты систем гипермедиа, поэтому мы рассмотрим его более подробно.

Ограничение рассмотрено в разделе 5.1.5[1].

В этом разделе Филдинг пишет:

> Центральной особенностью, отличающей архитектурный стиль REST от других сетевых стилей, становится особое внимание к унификации интерфейса между компонентами... Для формирования унифицированного интерфейса необходимы различные архитектурные ограничения, управляющие поведением компонентов. REST определяется четырьмя ограничениями интерфейса: идентификацией ресурсов; манипуляцией ресурсами через представления; самодокументируемыми сообщениями; и гипермедиа как ядром состояния приложения.
>
> *Рой Филдинг, «Architectural Styles and the Design of Network-based Software Architectures»*

Итак, имеются четыре подограничения, которые в совокупности формируют ограничение унифицированного интерфейса.

Идентификация ресурсов

В RESTful-системе ресурсы должны обладать уникальными идентификаторами. В наши дни концепция унифицированных указателей ресурсов, или URL, стала общеизвестной, но на момент написания диссертации Филдинга была относительно новой и революционной.

Сегодня большую важность имеет концепция самого *ресурса*, который должен быть идентифицируемым: в RESTful-системе *любая* разновидность данных, на которую можно ссылаться (то есть которая может быть целью гипермедиа-

[1] https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_1_5

ссылки), считается ресурсом. С помощью концепции URL, которая теперь используется повсеместно, удалось решить очень сложную проблему однозначной идентификации абсолютно любого ресурса в интернете.

Манипуляции с ресурсами через представления

В RESTful-системе *представления* ресурсов передаются между клиентами и серверами. Эти представления могут содержать как данные, так и метаданные, относящиеся к запросу (например, «управляющие данные», такие как метод HTTP или код ответа). Конкретный формат данных, или *тип информационной среды*, может использоваться для представления заданного ресурса клиенту, и этот тип информационной среды может быть согласован между клиентом и сервером.

Пример последнего мы видели в заголовке `Accept` из запросов выше.

Самодокументируемые сообщения

Ограничение самодокументируемости сообщений в сочетании со следующим ограничением HATEOAS образует то, что мы считаем ядром унифицированного интерфейса REST. Именно по этой причине гипермедиа предоставляет столь мощную системную архитектуру.

Ограничение самодокументируемости сообщения требует, чтобы в RESTful-системе сообщения *описывали сами себя*.

Это означает, что *вся информация*, необходимая как для отображения, так *и для выполнения операций* с представляемыми данными, должна присутствовать в ответе. В правильно спроектированной RESTful-системе не может быть дополнительной «побочной» информации, необходимой клиенту для преобразования ответа от сервера в полезный пользовательский интерфейс. Все должно содержаться в самом сообщении в форме гипермедиа-элементов управления.

Такое описание выглядит немного абстрактно, поэтому рассмотрим конкретный пример.

Возьмем два разных потенциальных ответа от сервера HTTP для URL https://example.com/contacts/42.

Оба ответа возвращают информацию о контакте, но воплощаются в совершенно разных формах.

Первая реализация возвращает представление HTML:

```
<html lang="en">
<body>
<h1>Joe Smith</h1>
<div>
```

```
  <div>Email: joe@example.bar</div>
  <div>Status: Active</div>
</div>
<p>
  <a href="/contacts/42/archive">Archive</a>
</p>
</body>
</html>
```

Вторая реализация возвращает представление JSON:

```
{
 "name": "Joe Smith",
 "email": "joe@example.org",
 "status": "Active"
}
```

Что можно сказать о различиях между двумя ответами?

Первое, что бросается в глаза, — представление JSON компактнее представления HTML. Филдинг обращает внимание именно на этот компромисс при использовании RESTful-архитектуры:

> “Однако расплачиваться приходится снижением эффективности унифицированного интерфейса, так как информация передается в стандартизированной форме, а не в той, которая адаптирована для потребностей приложения.
>
> *Рой Филдинг, «Architectural Styles and the Design of Network-based Software Architectures»*

Таким образом, REST *жертвует* эффективностью представления ради других целей.

Чтобы понять эти цели, для начала заметим, что представление HTML содержит гиперссылку для перехода к странице архивации контакта. А вот в представлении JSON такой ссылки нет.

Каковы последствия этого для *клиента* JSON API?

Это означает, что клиент JSON API должен *заранее* точно знать, какие другие URL (и методы запроса) доступны при работе с информацией контактов. Чтобы клиент JSON мог обновить данные контакта, он должен знать, как это сделать; следовательно, ему должен быть доступен некий источник информации, *внешний* по отношению к сообщению JSON. Если контакт имеет другой статус (например, Archived), изменит ли это допустимые действия? Если изменит, то какие еще действия будут разрешены?

Получить эту информацию можно из документации API, устно или, если под контролем разработчика находится как сервер, так и клиент, — исходя из знания внутреннего устройства. Но эта информация существует неявно и *за пределами* ответа.

Сравните с ответом гипермедиа (HTML). В таком случае гипермедиа-клиенту (то есть браузеру) необходимо знать только то, как отрендерить заданную разметку HTML. Ему не нужно понимать, какие действия доступны для этого контакта: они просто закодированы *внутри* самого ответа HTML как элементы управления. Ему не нужно понимать смысл поля `status`. Собственно, клиенту даже не нужно знать, что такое контакт!

Браузер, наш гипермедиа-клиент, просто рендерит HTML и дает возможность пользователю, который, как предполагается, понимает концепцию контакта, выбрать выполняемое действие из доступных в представлении.

Это различие между двумя ответами демонстрирует сущность REST и гипермедиа — то, что делает их такими мощными и гибкими: клиенты (также веб-браузеры) не обязаны *ничего* знать о представляемых ресурсах.

Браузеру необходимо понимать только (только! Как будто это просто!) то, как интерпретировать и отображать гипермедиа, в данном случае HTML. Это дает системам на основе гипермедиа беспрецедентную гибкость при обработке изменений как в используемых представлениях, так и в самой системе.

Гипермедиа как ядро состояния приложения (HATEOAS)

Последнее подограничение унифицированного интерфейса гласит, что в RESTful-системе гипермедиа является «ядром состояния приложения». Иногда используется сокращение HATEOAS (Hypermedia As The Engine of Application State), хотя Филдинг предпочитает термин «ограничение гипермедиа».

Это ограничение тесно связано с ограничением самодокументируемости сообщения. Снова рассмотрим две разные реализации конечной точки `/contacts/42`: одна возвращает HTML, а другая — JSON. Изменим ситуацию так, чтобы контакт, идентифицируемый URL, был архивирован.

Как будут выглядеть ответы?

Первая реализация возвращает следующую разметку HTML:

```
<html lang="en">
<body>
<h1>Joe Smith</h1>
<div>
  <div>Email: joe@example.bar</div>
  <div>Status: Archived</div>
</div>
```

```
<p>
  <a href="/contacts/42/unarchive">Unarchive</a>
</p>
</body>
</html>
```

Вторая реализация возвращает следующее представление JSON:

```
{
 "name": "Joe Smith",
 "email": "joe@example.org",
 "status": "Archived"
}
```

Важно отметить, что вследствие самодокументируемости сообщения из ответа HTML видно, что операция Archive (Архивировать) теперь недоступна, зато стала доступной новая операция Unarchive (Разархивировать). В представлении HTML контакта *закодировано* состояние приложения; оно точно определяет, что можно, а что нельзя сделать с этим конкретным представлением. Для представления JSON же такой возможности нет.

Еще раз: клиент, интерпретирующий ответ JSON, должен понимать не только общую концепцию контакта, но и конкретный смысл поля `status` со значением `Archived`. Он должен точно знать, какие операции доступны для контактов `Archived`, чтобы правильно отобразить их для конечного пользователя. Состояние приложения не кодируется в ответе, а передается через комбинацию низкоуровневых данных и сопутствующую информацию (например, документацию API).

Кроме того, в большинстве современных фронтенд-фреймворков SPA эта информация о контакте будет существовать *в памяти* в объекте JavaScript, представляющем модель контакта, тогда как данные страницы хранятся в модели *DOM*[1] (Document Object Model) браузера. DOM будет обновляться на основании изменений этой модели, то есть DOM будет «реагировать» на изменения поддерживающей модели JavaScript.

Разумеется, такой подход *не* использует концепцию HATEOAS; вместо этого ядром состояния приложения становится модель JavaScript, синхронизируемая с сервером и браузером.

В случае HTML ядром состояния приложения становится гипермедиа: на стороне клиента нет дополнительной модели, а все состояние выражается явно в гипермедиа, в данном случае HTML. Когда на сервере изменяется состояние, оно отражается в представлении (то есть HTML), возвращаемом клиенту. Кли-

[1] https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model

ент гипермедиа (браузер) ничего не знает о контактах, о концепции архивации контактов и вообще о конкретной модели предметной области для этого ответа; он просто знает, как рендерить HTML.

Так как клиенту гипермедиа не нужно ничего знать о модели сервера, достаточно только уметь визуализировать гипермедиа для клиента, он обладает большой гибкостью в отношении представлений, которые он получает и выводит для пользователей.

HATEOAS и пересмотр API

Последний пункт чрезвычайно важен для понимания гибкости гипермедиа, поэтому рассмотрим практический пример. Представим, что в веб-приложение добавлена новая функциональность с двумя конечными точками. Новая функциональность позволяет отправить сообщение заданному контакту.

Как при этом изменится каждый из двух ответов (HTML и JSON) сервера?

Обновленное представление HTML может выглядеть так:

```
<html lang="en">
<body>
<h1>Joe Smith</h1>
<div>
  <div>Email: joe@example.bar</div>
  <div>Status: Active</div>
</div>
<p>
  <a href="/contacts/42/archive">Archive</a>
  <a href="/contacts/42/message">Message</a>
</p>
</body>
</html>
```

С другой стороны, представление JSON может выглядеть так:

```
{
 "name": "Joe Smith",
 "email": "joe@example.org",
 "status": "Active"
}
```

Обратите внимание: представление JSON не изменилось. Ничто не указывает на появление новой функциональности. Клиент должен узнавать об этом изменении, вероятно, из некой общей документации для клиента и сервера.

Сравните это представление с ответом HTML. Благодаря унифицированному интерфейсу RESTful-модели и особенно тому, что мы используем HATEOAS

(гипермедиа как ядро состояния приложения), такой обмен документацией становится излишним! Клиент (браузер) просто рендерит новую разметку HTML с новой операцией, делая ее доступной для конечного пользователя без дополнительных изменений в коде.

Удобно, ничего не скажешь!

Если клиент JSON корректно не обновить, в данном примере ошибка будет относительно безобидной: новая функциональность окажется просто недоступной пользователям. Но представьте более серьезное изменение API: например, удаление функции архивации? Или изменение URL или методов HTTP этих операций?

В таком случае нарушения в работе клиента JSON могут оказаться куда более серьезными.

Ответ HTML же будет просто обновлен: из него будут исключены лишние варианты, или изменятся используемые в них URL. Клиент получит новую разметку HTML, правильно отобразит ее, и у пользователя будет возможность выбрать операции из нового набора. И снова унифицированный интерфейс REST оказывается чрезвычайно гибким: несмотря на потенциальные серьезные изменения в API гипермедиа, клиенты продолжают работать.

Из этого следует один важный факт: *благодаря гибкости у API гипермедиа не будет проблем с контролем версий, присущих API данных JSON.*

После «входа» в гипермедиа-управляемое приложение (то есть после загрузки по стартовому URL) вся функциональность и ресурсы становятся доступными через самодокументируемые сообщения. Следовательно, обмениваться документацией с клиентом уже не нужно: клиент просто рендерит гипермедиа (в данном случае HTML), и все работает. Когда происходит изменение, создавать новую версию API не обязательно: клиент просто получает обновленный вариант гипермедиа, в котором закодированы новые операции и ресурсы, и предоставляет его пользователям для взаимодействия.

Многоуровневые системы

Последнее «обязательное» ограничение для RESTful-систем — ограничение многоуровневой системы. Его можно найти в разделе 5.1.6[1] диссертации Филдинга.

Откровенно говоря, после интересного и нетривиального ограничения унифицированного интерфейса ограничение «многоуровневой системы» немно-

[1] https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_1_6

го разочаровывает. Но его все равно стоит понимать, и на самом деле оно эффективно используется в веб-среде. Ограничение требует, чтобы RESTful-архитектура была «многоуровневой», в которой несколько серверов могут действовать как посредники между клиентом и сервером — конечным «источником истины». Промежуточные серверы выполняют функции прокси-серверов, осуществляют промежуточные преобразования запросов и ответов и т. д.

Типичный современный пример использования многоуровневой структуры REST — применение *сетей доставки контента* (*CDN*, Content Delivery Network) для ускорения доставки неизменяемых статических ресурсов клиентам за счет хранения ответа сервера-источника на промежуточных серверах, которые расположены ближе к клиенту, выдающему запрос.

Это позволяет ускорить доставку контента конечному пользователю и сократить нагрузку на сервер-источник.

Данное ограничение не настолько интересно для разработчиков веб-приложений, как ограничение унифицированного интерфейса (по крайней мере, на наш взгляд). Тем не менее оно имеет практическую пользу.

Необязательное ограничение: Code-On-Demand

Мы назвали ограничение многоуровневой системы последним «обязательным», потому что Филдинг упоминает еще одно дополнительное ограничение для RESTful-систем. Ограничение Code-on-Demand несколько грубо описывается как «необязательное» (раздел 5.1.7).

В этом разделе Филдинг пишет:

> “ REST позволяет расширять клиентскую функциональность посредством загрузки и выполнения кода в форме апплетов или скриптов. Данная возможность упрощает клиент за счет сокращения количества функций, которые необходимо реализовать заранее. Возможность загрузки дополнительной функциональности после развертывания улучшает расширяемость системы. Однако при этом сокращается видимость изменений, поэтому в REST это ограничение считается необязательным.
>
> *Рой Филдинг, «Architectural Styles and the Design of Network-based Software Architectures»*

Таким образом, скрипты были и остаются встроенным компонентом исходной RESTful-модели веб-среды, и конечно, они должны поддерживаться в гипермедиа-управляемых приложениях.

Однако присутствие скриптовых технологий *не должно* изменять фундаментальную модель сетевых взаимодействий в гипермедиа-управляемых приложениях: гипермедиа остается ядром состояния приложения, взаимодействия с сервером состоят из обмена информацией гипермедиа (вместо, например, обмена данными JSON) и т. д. (Разумеется, в приложении останется место для интерфейсов JSON Data API; в главе 10 мы обсудим, когда и как с ними работать.)

К сожалению, в наши дни скриптовый уровень веб-среды — JavaScript — часто используется как *замена*, а не как дополнение модели гипермедиа. В одной из следующих глав будет показано, как выглядят скриптовые технологии, не заменяющие базовую гипермедиа-систему веб-среды.

Заключение

После такого подробного знакомства с компонентами и концепциями, лежащими в основе систем гипермедиа, включая представления Роя Филдинга о принципах их работы, мы надеемся, что вы стали намного лучше понимать REST, и в частности унифицированный интерфейс и HATEOAS. Надеемся, теперь вам ясно, *почему* благодаря этим особенностям системы гипермедиа обладают столь выдающейся гибкостью.

Если до сих пор вы не в полной мере представляли всю значимость REST и HATEOAS, не огорчайтесь: для осознания особой природы HTML, гипермедиа и веб-среды нам потребовалось почти десять лет провести в веб-разработке и построить гипермедиа-ориентированную библиотеку с нуля!

Заметки об HTML: каша из HTML5

> “Мудрость начинается с того, чтобы называть вещи подходящими именами.
>
> *Конфуций*

Такие элементы, как `<section>`, `<article>`, `<nav>`, `<header>`, `<footer>`, `<figure>` в HTML стали своего рода сокращением.

Используя эти элементы, страница может давать легковерным клиентам — браузерам, поисковым системам и ботам — ложные обещания (например, что элементы `<article>` представляют автономные сущности, пригодные для повторного использования). Чтобы этого не происходило:

- следите, чтобы выбранный элемент соответствовал сценарию использования. Обращайтесь к спецификации HTML;

- не пытайтесь выражать намерения конкретно, если это невозможно или необязательно. Иногда использование `<div>` оправданно.

Самый авторитетный ресурс по HTML — спецификация HTML. Актуальная спецификация находится по адресу https://html.spec.whatwg.org/multipage[1]. Чтобы оставаться в курсе изменений в HTML, не стоит полагаться на слухи.

В разделе 4 спецификации приведен список всех доступных элементов с указанием, что они представляют, где могут встречаться, что могут содержать и т. д. В нем даже указано, в каких случаях можно опускать закрывающие теги!

[1] Одностраничная версия слишком долго загружается и отображается на большинстве компьютеров. Помимо нее существует версия для разработчиков в /dev, но в стандартной версии используется более приятное стилевое оформление.

ПРИЛОЖЕНИЕ WEB 1.0

Начнем путешествие в мир гипермедиа-управляемых приложений с создания простого веб-приложения для управления контактами, которое будет называться Contact.app. Это будет базовое многостраничное приложение (MPA) «в стиле Web 1.0», использующее традиционную схему CRUD. Возможно, это не лучшее в мире приложение для управления контактами, но оно простое и рабочее.

В дальнейших главах мы будем постепенно улучшать созданное приложение при помощи гипермедиа-ориентированной библиотеки `htmx`.

К тому моменту, когда мы завершим построение и доработку приложения, в нем будет реализован ряд классных возможностей, для которых, по мнению многих сегодняшних разработчиков, необходимо использовать фреймворк SPA с JavaScript.

Выбор веб-стека

Чтобы продемонстрировать, как работают приложения Web 1.0, необходимо выбрать язык на стороне сервера и библиотеку для обработки запросов HTTP. В совокупности это называется «стеком на стороне сервера», или «веб-стеком». Существуют сотни их видов, и многие имеют своих убежденных приверженцев. Скорее всего, у вас есть веб-фреймворк, который вы предпочитаете другим. И хотя мы хотели бы включить в эту книгу описания всех возможных стеков, ради простоты (и для сохранения рассудка) придется выбрать что-то одно.

В этой книге будет использоваться следующий стек:

- Python[1] — язык программирования.
- Flask[2] — веб-фреймворк, позволяющий связывать запросы HTTP с логикой Python.
- Jinja2[3] — язык шаблонов на стороне сервера, позволяющий рендерить ответы HTML с использованием знакомого и интуитивно понятного синтаксиса.

Почему именно этот стек?

---

[1] https://www.python.org/

[2] https://palletsprojects.com/p/flask/

[3] https://palletsprojects.com/p/jinja/

На момент написания книги Python был самым популярным в мире языком программирования согласно индексу TIOBE[1] — авторитетному источнику информации о популярности языков программирования. Что еще важнее, код Python легко читать, даже если вы не изучали этот язык.

Мы выбрали веб-фреймворк Flask за его простоту, а также потому, что он не накладывает заметную дополнительную структуру поверх основной функциональности обработки запросов HTTP.

Этот минималистский подход хорошо годится для наших целей; в других случаях можно подумать об использовании более полнофункционального фреймворка Python (такого, как Django[2]), предоставляющего намного больше готовой функциональности, чем Flask.

Использование Flask в книге позволяет сосредоточиться в коде на *гипермедийном обмене данными*.

Мы выбрали шаблоны Jinja2, потому что они являются шаблонами по умолчанию для Flask. Они достаточно просты и похожи на другие языки шаблонов на стороне сервера, поэтому их быстро и легко поймут большинство разработчиков, знакомых с любой библиотекой шаблонов на стороне сервера (или на стороне клиента).

Даже если вы предпочитаете другую комбинацию технологий, продолжайте читать: вы узнаете немало полезного из паттернов, которые будут представлены в следующих главах, и сможете без особого труда связать их со своим любимым языком и фреймворком.

С этим стеком мы будем рендерить *на стороне сервера* разметку HTML, возвращаемую клиентам, а не JSON. Так выглядел традиционный подход к построению веб-приложений. Однако с ростом популярности SPA он перестал использоваться так же широко, как прежде. В наши дни, когда разработчики заново открывают этот стиль построения веб-приложений, в его обсуждениях стал звучать термин «рендеринг на стороне сервера», или SSR (Server Side Rendering). Он противопоставляется «рендерингу на стороне клиента», то есть визуализации шаблонов в браузере данными, полученными в форме JSON от сервера, как это обычно делается в библиотеках SPA.

В приложении Contact.app мы намеренно постарались упростить задачу, чтобы повысить учебную ценность кода: структура этого кода неидеальна, зато читателю будет просто разобраться в его логике даже с небольшим опытом работы на Python и он сможет преобразовать приложение и продемонстрированные методы в ту среду программирования, которую предпочитает.

---

[1] https://www.tiobe.com/tiobe-index/

[2] https://www.djangoproject.com/

Python

Так как эта книга учит эффективно использовать гипермедиа, стоит хотя бы кратко упомянуть технологии, используемые *вместе с* гипермедиа. У такого подхода имеются очевидные недостатки: например, если вы не владеете Python, некоторые примеры кода Python в книге на первый взгляд могут показаться непонятными или запутанными.

Если вы чувствуете, что перед погружением в код вам понадобится краткий вводный курс языка, мы рекомендуем следующие книги и сайты:

- Python Crash Course (No Starch Press)[1];
- Learn Python The Hard Way (Zed Shaw)[2];
- Python For Everybody (Dr. Charles R. Severance)[3].

Мы считаем, что большинство веб-разработчиков, даже незнакомых с Python, смогут понять логику примеров. Многие авторы не писали код Python до этой книги, но освоились довольно быстро.

Знакомство с Flask: первый маршрут

Flask — простой, но гибкий веб-фреймворк для Python. Чтобы дать вам начальное представление о нем, мы рассмотрим его базовые элементы.

Приложение Flask состоит из серии *маршрутов*, связанных с функциями, выполняемыми при выдаче запроса HTTP к соответствующему пути. В нем используется так называемая функция-декоратор Python для объявления обрабатываемого маршрута, за которой следует функция для обработки запросов к этому маршруту. Мы будем называть функции, связанные с маршрутом, обработчиками (handler).

Создадим свое первое определение маршрута — простой маршрут Hello World. В следующем коде Python встречается обозначение `@app`. Это декоратор Flask, который позволяет нам определять собственные маршруты. Пока не думайте о том, как работают декораторы в Python; просто знайте, что они позволяют связать заданный *путь* с конкретной функцией (например, обработчиком). Приложение Flask при запуске получает запросы HTTP, ищет подходящий обработчик и вызывает его.

---

[1] Мэтиз Э. Изучаем Python. СПб.: издательство «Питер».

[2] Шоу З. Легкий способ выучить Python 3.

[3] Северанс Ч. Python для всех.

Листинг 10. Простой маршрут Hello World

```
@app.route("/") ❶
def index():  ❷
  return "Hello World!" ❸
```

❶ Определяет, что путь / отображается на маршрут.
❷ Следующий метод является обработчиком для этого маршрута.
❸ Возвращает клиенту строку «Hello World!».

Метод `route()` декоратора Flask получает аргумент: путь, который должен обрабатываться маршрутом. Здесь корневой путь (или путь `/`) передается в строковом виде для обработки запросов к корневому маршруту.

За объявлением маршрута следует простое определение функции `index()`. В Python декораторы, вызываемые таким образом, применяются к функции, следующей непосредственно за ними. Таким образом, эта функция становится «обработчиком» для указанного маршрута и будет выполняться при выдаче запроса HTTP к заданному пути.

Учтите, что имя функции роли не играет; ее можно назвать как угодно. Требуется лишь, чтобы имя было уникальным. В данном случае мы выбрали имя `index()`, потому что оно соответствует обрабатываемому маршруту: корню index веб-приложения.

Итак, мы размещаем функцию `index()` сразу же за определением маршрута для корня, и эта функция становится обработчиком для корневого URL в веб-приложении.

Обработчик в данном случае невероятно прост — он всего лишь возвращает клиенту строку «Hello World!». Это еще не гипермедиа, но браузер корректно отображает эту строку.

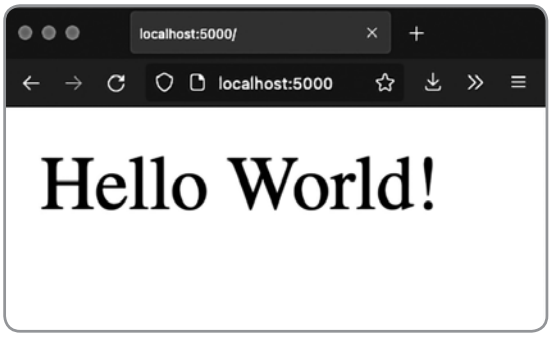


Рис. 3. Hello World!

Итак, наш первый шаг в мире Flask демонстрирует основной прием, который будет использоваться для ответа на запросы HTTP: связывание маршрутов с обработчиками.

В приложении Contact.app вместо того, чтобы выдавать строку «Hello World!» по корневому пути, мы сделаем кое-что более интересное: перенаправление по другому пути, `/contacts`. Перенаправление относится к возможностям HTTP, позволяющим перенаправить клиент в другое место при помощи ответа HTTP.

На корневой странице будет выводиться список контактов, и, пожалуй, перенаправление по пути `/contacts` для вывода этой информации чуть лучше соответствует концепции ресурсов в REST. С нашей стороны это субъективное решение; может быть, оно не принципиально важно, но выглядит разумно применительно к маршрутам, которые будут создаваться далее в приложении.

Чтобы заменить маршрут Hello World на перенаправление, достаточно изменить всего одну строку кода.

Листинг 11. Преобразование Hello World в перенаправление

```
@app.route("/")
def index():
  return redirect("/contacts") ❶
```

❶ Заменяется вызовом redirect().

Теперь функция `index()` возвращает результат функции `redirect()`, предоставленной Flask, с переданным при вызове путем. В данном случае используется путь `/contacts`, передаваемый в строковом аргументе. Теперь при переходе по корневому пути `/` наше приложение Flask перенаправляет пользователя на путь `/contacts`.

Функциональность Contact.app

Итак, вы в общих чертах понимаете, как определять маршруты, и мы можем перейти к определению и реализации веб-приложения.

Что будет делать приложение Contact.app?

В исходной версии пользователь сможет:

- просмотреть список контактов, включая имя, фамилию, номер телефона и адрес электронной почты;
- найти нужный контакт;
- добавить новый контакт;
- просмотреть подробную информацию о контакте;

- отредактировать подробную информацию о контакте;
- удалить контакт.

Как видите, Contact.app является приложением CRUD, идеально подходящим для традиционного подхода Web 1.0.

Отметим, что исходный код Contact.app доступен на GitHub[1].

Вывод списка контактов с возможностью поиска

Добавим первую часть реальной функциональности: возможность вывода всех контактов в приложении в списке (а на самом деле в таблице).

Эта функциональность будет доступна на пути `/contacts`, по которому осуществлял перенаправление маршрут выше.

Мы воспользуемся Flask для связывания пути `/contacts` с функцией-обработчиком `contacts()`. Эта функция будет делать одно из двух:

- если в запросе присутствует условие поиска — фильтровать список до контактов, соответствующих этому условию;
- если условия поиска нет — просто выводить все контакты списком.

Это типичный способ организации поведения приложений в стиле Web 1.0: URL, выводящий все экземпляры некоторого ресурса, также служит страницей результатов поиска для этих ресурсов. Он упрощает повторное использование функциональности вывода списка, общей для обоих видов запросов.

Код обработчика выглядит примерно так:

Листинг 12. Обработчик поиска на стороне сервера

```
@app.route("/contacts")
def contacts():
  search = request.args.get("q") ❶
  if search is not None:
    contacts_set = Contact.search(search) ❷
  else:
    contacts_set = Contact.all() ❸
  return render_template("index.html", contacts=contacts_set) ❹
```

❶ Ищет параметр запроса с именем q (сокращение от «query»).
❷ Если параметр существует, вызывает функцию Contact.search() с передачей параметра.
❸ Если параметра не существует, вызывает функцию Contact.all().
❹ Передает результат шаблону index.html для визуализации на стороне клиента.

[1] https://github.com/bigskysoftware/contact-app

Мы видим такой же код маршрутизации, как в первом примере, но с более сложной функцией-обработчиком. Сначала мы проверяем, присутствует ли в запросе параметр с именем `q`.

Строки запросов

> «Строка запроса» является частью спецификации URL. Пример URL со строкой запроса: https://example.com/contacts?q=joe. Строка запроса содержит все символы после `?` и кодируется в формате пар «имя — значение». В этом URL параметру запроса `q` присваивается строковое значение `joe`. В простой разметке HTML строка запроса либо жестко кодируется в якорном теге, либо, в более динамическом варианте, в нее включается тег формы с запросом `GET`.

Вернемся к маршруту Flask: если параметр запроса с именем `q` найден, то приложение вызывает метод `search()` для объекта модели `Contact`, чтобы выполнить поиск контакта и вернуть все подходящие контакты.

Если параметр запроса *не* найден, мы просто получаем все контакты, вызывая метод `all()` для объекта `Contact`.

Наконец, приложение рендерит шаблон `index.html`, который отображает заданные контакты, с передачей результатов одной из двух функций, которая была вызвана ранее.

О КЛАССЕ CONTACT

Класс Python `Contact`, который мы используем, является классом «модели предметной области», или просто «модели», нашего приложения. Он предоставляет «бизнес-логику», связанную с управлением контактами.

Он может работать с базой данных (но этого не делает) или простым неструктурированным файлом (и делает это), но мы пропустим внутренние подробности модели. Считайте, что перед вами «обычный» класс предметной области с методами, которые работают «как обычно».

Мы будем считать `Contact` *ресурсом* и основное внимание уделим тому, как эффективно передать гипермедиа-представления этого ресурса клиентам.

Шаблоны вывода списка и результатов поиска

Итак, логика обработчика написана, и мы можем создать шаблон, который рендерит HTML для ответа клиенту. На высоком уровне ответ HTML должен включать следующие элементы:

- список всех контактов или контактов, соответствующих условию поиска;

- поле поиска, в котором пользователь может ввести и отправить условие поиска;
- несколько дополнительных элементов интерфейса: «шапку» и «подвал» сайта, которые остаются одинаковыми независимо от того, на какой странице вы находитесь.

Мы используем язык шаблонов Jinja2, который предоставляет следующие возможности:

- двойные фигурные скобки `{{ }}` предназначены для встраивания значений выражений в шаблон;
- фигурные скобки со знаком процента `{% %}` используются для директив, например перебора или включения другого контента.

Помимо базового синтаксиса, язык Jinja2 очень похож на другие языки шаблонов, используемые для генерирования контента, и большинство веб-разработчиков разберутся в нем без особых проблем.

Посмотрите на начальные строки кода шаблона `index.html`.

Листинг 13. Начало шаблона index.html

```
{% extends 'layout.html' %} ❶

{% block content %} ❷

  <form action="/contacts" method="get" class="tool-bar"> ❸
      <label for="search">Search Term</label>
      <input id="search" type="search" name="q" value="{{
request.args.get('q') or '' }}"/> ❹
      <input type="submit" value="Search"/>
  </form>
```

❶ Назначает базовый шаблон макета layout.html.
❷ Определяет границы контента, вставляемого в макет.
❸ Создает форму поиска, которая выдает запрос HTTP GET для /contacts.
❹ Создает поле ввода поисковых запросов.

В первой строке кода назначается базовый шаблон макета layout.html, для чего используется директива `extends`. Шаблон макета определяет макет страницы (также иногда называемый «хромом», «chrome»): он заключает контент шаблона в тег `<html>`, импортирует всю необходимую разметку CSS и код JavaScript в элементе `<head>`, размещает тег `<body>` вокруг основного контента и т. д. В этом файле находится весь общий контент, в который упаковывается «обычный» контент всего приложения.

В следующей строке кода объявляется раздел `content` этого шаблона. Блок контента используется шаблоном `layout.html` для внедрения содержимого `index.html` в свою разметку HTML.

Далее идет первый блок «настоящей» разметки HTML, в отличие от простых директив Jinja. Простая форма HTML позволяет проводить поиск контактов с выдачей запроса `GET` к пути `/contacts`. Сама форма содержит метку и поле ввода с именем `"q"`. Значение этого поля отправляется с запросом `GET` к пути `/contacts` в виде строки запроса (так как используется запрос `GET`).

Обратите внимание: в качестве значения поля ввода указывается выражение Jinja `{{request.args.get('q') or '' }}`. Jinja оценивает это выражение, и если оно существует, вставляет значение запроса `q` как значение поля ввода. Таким образом, искомое значение «сохраняется» при выполнении поиска, чтобы при отображении результатов поиска текстовое поле содержало запрос, по которому велся поиск. Это улучшает опыт взаимодействия с пользователем, так как пользователь видит, чему соответствуют текущие результаты поиска, а не просто пустое текстовое поле в верхней части экрана.

Разметка завершается полем ввода с типом `submit`. Оно рендерится в виде кнопки, по щелчку которой форма отправляет запрос HTTP.

Интерфейс поиска занимает верхнюю часть страницы контактов. Ниже идет таблица с контактами — либо со всеми, либо с соответствующими условиям поиска, если выполнялся поиск.

Вот как выглядит код шаблона для таблицы контактов.

Листинг 14. Таблица контактов

```
  <table>
    <thead>
    <tr>
      <th>First</th> <th>Last</th> <th>Phone</th> <th>Email</th> <th>
</th> ❶
    </tr>
    </thead>
    <tbody>
    {% for contact in contacts %} ❷
      <tr>
        <td>{{ contact.first }}</td>
        <td>{{ contact.last }}</td>
        <td>{{ contact.phone }}</td>
        <td>{{ contact.email }}</td> ❸
        <td><a href="/contacts/{{ contact.id }}/edit">Edit</a>
          <a href="/contacts/{{ contact.id }}">View</a></td> ❹
      </tr>
```

```
    {% endfor %}
    </tbody>
  </table>
```

❶ Выводит заголовки для таблицы.
❷ Перебирает контакты, переданные шаблону.
❸ Выводит значения текущего контакта: имя, фамилия и т. д.
❹ Столбец «операций» со ссылками для редактирования или просмотра подробной информации контакта.

Это ядро страницы: мы строим таблицу с заголовками, соответствующими данным, которые должны отображаться для каждого контакта. Мы перебираем контакты, переданные шаблону методом-обработчиком с использованием директивы цикла `for` в Jinja2. Затем формируем последовательность строк, по одной для каждого контакта, в ячейках которых выводятся имя и фамилия, телефон и электронная почта контакта.

Также имеется ячейка таблицы, включающая две ссылки:

- Ссылка на страницу Edit (Редактирование) для контакта по маршруту `/contacts/{{ contact.id }}/edit` (например, для контакта с идентификатором 42 ссылка на редактирование указывает на `/contacts/42/edit`).
- Ссылка на страницу View (Просмотр) для контакта `/contacts/{{contact.id }}` (в приведенном выше примере страница просмотра будет иметь вид `/contacts/42`).

Ниже расположены завершающие элементы: ссылка для добавления нового контакта и директива Jinja2 для завершения блока `content`.

Листинг 15. Ссылка Add contact для добавления нового контакта

```
  <p>
    <a href="/contacts/new">Add Contact</a> ❶
  </p>

{% endblock %} ❷
```

❶ Ссылка на страницу для создания нового контакта.
❷ Закрывающий элемент блока content.

Вот и весь шаблон. Используя этот простой шаблон на стороне сервера в сочетании с методом-обработчиком, мы можем отвечать на запрос *представлением* HTML всех запрашиваемых контактов. Пока все гипермедийно.

А вот как выглядит шаблон, отрендеренный с данными нескольких контактов.

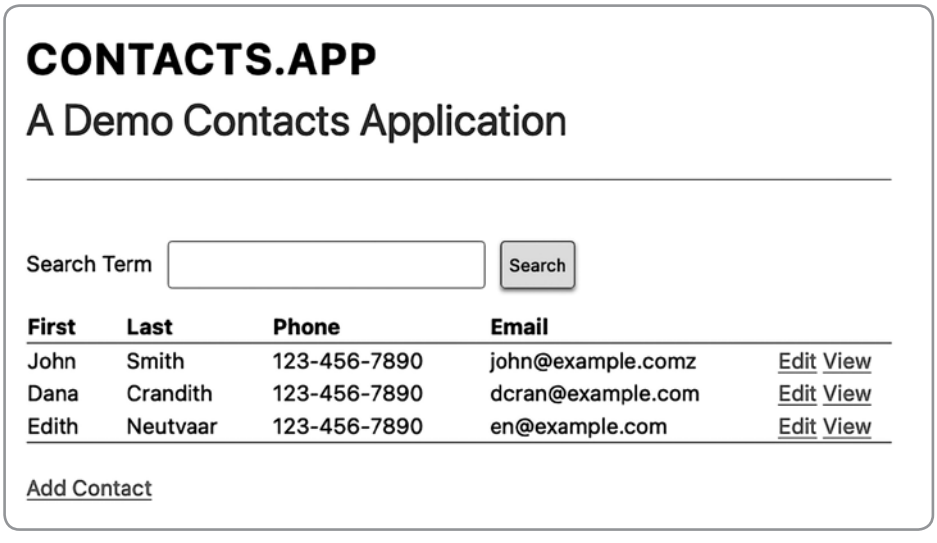


Рис. 4. Contact.app

Такое приложение вряд ли получит приз за лучший дизайн, но стоит заметить, что наш шаблон после рендеринга предоставляет всю функциональность, необходимую для просмотра контактов и их поиска, а также содержит ссылки для редактирования, просмотра подробной информации и даже создания новых контактов.

И при этом клиенту (то есть браузеру) не нужно ничего знать о том, что собой представляют контакты или как работать с ними. Все закодировано в гипермедиа. Веб-браузер, обращающийся к приложению, просто умеет выдавать запросы HTTP и рендерить HTML — он ничего не знает о специфике конечных точек приложения или используемой модели предметной области.

Каким бы простым ни было приложение на данном этапе, оно полностью соответствует принципам REST.

Добавление нового контакта

Создадим в приложении возможность добавления новых контактов. Для этого нужно будет обработать URL `/contacts/new`, доступный по представленной выше ссылке Add Contact. Обратите внимание: когда пользователь щелкает на этой ссылке, браузер выдает запрос `GET` к URL `/contacts/new`.

Все остальные маршруты, которые были определены ранее, также используют `GET`, но в реализации новой функциональности будут применяться два разных метода HTTP: HTTP `GET` для рендеринга формы добавления нового контакта

и HTTP POST *к тому же пути* для фактического создания контакта. А значит, при объявлении маршрута необходимо явно указать, какой метод HTTP он должен обрабатывать.

Листинг 16. Маршрут GET для нового контакта

```
@app.route("/contacts/new", methods=['GET']) ❶
def contacts_new_get():
  return render_template("new.html", contact=Contact()) ❷
```

❶ Объявление маршрута, явно обрабатывающего запросы GET к этому пути.
❷ Рендеринг шаблона new.html с передачей нового объекта контакта.

Ничего сложного. Мы просто рендерим шаблон `new.html` с новым экземпляром `Contact`. (`Contact()` — способ построения нового экземпляра класса `Contact` в Python, если вдруг вы не встречались с этим синтаксисом.)

Код обработчика для этого маршрута очень прост, но шаблон `new.html` получается более сложным.

> В остальных шаблонах мы будем опускать директиву layout и объявление блока контента, но вы можете предполагать, что они остаются неизменными, если только не указано обратное. Это позволит нам сосредоточиться на сути шаблона.

Если вы знакомы с HTML, вероятно, вы ожидаете увидеть элемент `form` — и эти ожидания вполне оправданны. Мы воспользуемся стандартным элементом гипермедиа `form` для сбора контактной информации и отправки ее на сервер.

Разметка HTML будет выглядеть так:

Листинг 17. Форма для создания контакта

```
<form action="/contacts/new" method="post"> ❶
  <fieldset>
    <legend>Contact Values</legend>
    <p>
      <label for="email">Email</label> ❷
      <input name="email" id="email" type="email" placeholder="Email"
value="{{ contact.email or '' }}"> ❸
      <span class="error">{{ contact.errors['email'] }}</span> ❹
    </p>
```

❶ Форма, отправляющая данные в /contacts/new с использованием HTTP POST.
❷ Метка для первого поля ввода формы.

❸ Первое поле ввода формы с типом email.
❹ Сообщения об ошибках, связанные с этим полем.

В первой строке кода создается форма, которая отправляет данные обратно *к тому же пути*, который мы обрабатываем: `/contacts/new`. Однако вместо того, чтобы выдавать запрос HTTP `GET` к этому пути, мы выдаем запрос HTTP `POST`. Такое использование `POST` сигнализирует серверу о том, что требуется создать новый контакт, а не получать форму для его создания.

Далее следуют метка (метки, поясняющие смысл полей, — это хорошая практика!) и поле ввода для адреса электронной почты создаваемого контакта. Полю ввода назначается имя `email`, и при отправке данных формы значение этого поля, ассоциированное с ключом `email`, отправляется в запросе `POST`.

Далее следуют поля ввода для остальных атрибутов контакта.

Листинг 18. Поля ввода и метки формы для создания контакта

```
    <p>
      <label for="first_name">First Name</label>
      <input name="first_name" id="first_name" type="text"
placeholder="First Name" value="{{ contact.first or '' }}">
      <span class="error">{{ contact.errors['first'] }}</span>
    </p>
    <p>
      <label for="last_name">Last Name</label>
      <input name="last_name" id="last_name" type="text"
placeholder="Last Name" value="{{ contact.last or '' }}">
      <span class="error">{{ contact.errors['last'] }}</span>
    </p>
    <p>
      <label for="phone">Phone</label>
      <input name="phone" id="phone" type="text" placeholder="Phone"
value="{{ contact.phone or '' }}">
      <span class="error">{{ contact.errors['phone'] }}</span>
    </p>
```

В конце помещаются кнопка отправки данных формы, завершающий тег `form` и обратная ссылка на основную таблицу контактов.

Листинг 19. Кнопка отправки данных формы для создания контакта

```
    <button>Save</button>
  </fieldset>
</form>

<p>
  <a href="/contacts">Back</a>
</p>
```

В этом простом примере легко упустить из виду один факт: он показывает всю гибкость гипермедиа в действии.

При добавлении или удалении поля, изменении логики проверки полей или их взаимодействий новое состояние будет отражено в новом представлении гипермедиа, передаваемом пользователю. Пользователь видит обновленную форму и может работать с новой функциональностью, для чего не потребуется обновлять приложение.

Обработка запроса POST к /contacts/new

Следующим нашим шагом станет обработка запроса `POST`, который отправляется формой к `/contacts/new`.

Для этого необходимо добавить в приложение новый маршрут, для пути `/contacts/new`. Новый маршрут будет обрабатывать метод HTTP `POST` вместо HTTP `GET`. Значения отправленной формы будут использоваться для создания нового экземпляра `Contact`.

Если экземпляр `Contact` создан успешно, пользователь перенаправляется к списку контактов и выводится сообщение об успехе. Если же попытка создания завершилась неудачей, приложение снова отображает форму создания контакта со значениями, введенными пользователем, и выдает сообщение об ошибках, которые должен исправить пользователь.

Новый обработчик запроса выглядит так:

Листинг 20. Контроллер формы для создания контакта

```
@app.route("/contacts/new", methods=['POST'])
def contacts_new():
  c = Contact(None, request.form['first_name'], request.form['last_name'],
request.form['phone'],
        request.form['email']) ❶
  if c.save(): ❷
    flash("Created New Contact!")
    return redirect("/contacts") ❸
  else:
    return render_template("new.html", contact=c) ❹
```

❶ Конструирует новый объект Contact со значениями из формы.

❷ Пытается сохранить его.

❸ В случае успеха выводится флеш-сообщение об успешном создании нового контакта («Created new contact!») и происходит перенаправление на страницу /contacts.

❹ В случае неудачи форма выводится заново с сообщением об ошибках для пользователя.

Логика этого обработчика немного сложнее других рассмотренных ранее методов. Все начинается с создания нового экземпляра `Contact`, при этом для кон-

струирования объекта снова используется синтаксис `Contact()` языка Python. Для передачи значений, отправленных пользователем с формой, используется объект `request.form` — эту возможность предоставляет Flask.

Объект `request.form` позволяет легко и удобно обращаться к значениям отправленной формы. Для этого значения передаются с именами, связанными с различными полями ввода формы.

Также в первом значении конструктора `Contact` передается `None`. В этом параметре должен передаваться идентификатор. Передавая `None`, мы сигнализируем, что это новый контакт, для которого необходимо сгенерировать идентификатор. (Напомним, что мы не будем вдаваться в подробности реализации объекта модели и нас интересует только его использование для выдачи ответов гипермедиа.)

Затем вызывается метод `save()` для объекта `Contact`. Этот метод возвращает `true`, если сохранение прошло успешно, или `false`, если попытка сохранения привела к ошибке (например, пользователь указал некорректный адрес электронной почты).

Если контакт был сохранен (то есть ошибки проверки данных отсутствуют), приложение создает *флеш-сообщение* об успехе, а браузер перенаправляется обратно на страницу списка. «Флеш-сообщения» — стандартная функция веб-фреймворков, позволяющая сохранить сообщение (как правило, в cookie или сеансовом хранилище), которое будет доступно в следующем запросе.

Наконец, если сохранить контакт не удалось, мы заново рендерим шаблон `new.html` с контактом. Шаблон останется прежним, но поля ввода уже будут заполнены отправленными ранее значениями и в качестве обратной связи для пользователя, объясняющей, почему данные не прошли проверку, будут выведены все ошибки, совершенные при вводе.

ПАТТЕРН POST/ REDIRECT/GET

В этом обработчике реализована стандартная стратегия разработки в стиле Web 1.0 – так называемый паттерн Post/Redirect/Get[1], или PRG. Выдавая перенаправление HTTP после создания контакта и указывая браузеру другое местоположение, мы гарантируем, что запрос POST не окажется в кэше запросов браузеров.

Это означает, что если пользователь случайно (или намеренно) обновит страницу, браузер не выдаст еще один запрос POST, который мог бы привести к созданию еще одного контакта. Вместо этого он выдает запрос GET, к которому выполняется перенаправление, свободное от побочных эффектов.

Паттерн PRG будет использоваться в книге неоднократно.

[1] https://en.wikipedia.org/wiki/Post/Redirect/Get

Итак, мы подготовили логику сохранения контактов на стороне сервера. И хотите верьте, хотите нет, но другие наши обработчики будут столь же простыми, как этот, даже когда мы начнем добавлять более сложное поведение, управляемое `htmx`.

Просмотр подробной информации о контакте

Следующая функциональность, которую мы реализуем, — страница подробной информации о контакте. Чтобы перейти к этой странице, пользователь должен щелкнуть на ссылке `View` в одной из строк списка контактов. Ссылка направляет пользователя по пути `/contact/<contact_id>` (например, `/contacts/42`). Это типичный паттерн веб-разработки: контакты рассматриваются как ресурсы, а URL этих ресурсов последовательно организованы.

- Если вы хотите просмотреть все контакты, выдайте запрос `GET` к `/contacts`.
- Если вам нужно гипермедиа-представление, позволяющее создавать новые контакты, выдайте запрос `GET` к `/contacts/new`.
- Если вы хотите просмотреть конкретный контакт (допустим, с идентификатором 42), выдайте запрос `GET` к `/contacts/42`.

ВЕЧНЫЕ ПРОБЛЕМЫ СО СТРУКТУРОЙ URL

Найти недостатки можно в любой схеме путей, используемой в приложении:

«Отправлять запрос POST лучше к /contacts/new или к /contacts?»

В Сети ведется множество споров на эту тему, и оба подхода имеют своих сторонников. На наш взгляд, важнее понимать общую идею *ресурсов* и *гипермедиа-представлений*, а не вязнуть в мелких подробностях структуры URL.

Мы рекомендуем выбрать разумную ресурсно-ориентированную структуру URL по своему вкусу и затем ее придерживаться. Помните, что в системе гипермедиа изменить конечные точки можно в любой момент, ведь вы используете гипермедиа как ядро состояния приложения!

Логика обработчика для маршрута подробной информации будет *очень* простой: мы будем искать `Contact` по идентификатору, встроенному в путь URL маршрута. Чтобы извлечь идентификатор, добавим еще функциональности Flask: выделение компонентов пути, их автоматическое извлечение и передачу функции-обработчику.

Вот как выглядит код, состоящий всего из нескольких простых строк на Python:

```
@app.route("/contacts/<contact_id>") ❶
def contacts_view(contact_id=0): ❷
  contact = Contact.find(contact_id) ❸
  return render_template("show.html", contact=contact) ❹
```

❶ Регистрирует путь с переменной с именем contact_id.
❷ Обработчик получает значение параметра этого пути.
❸ Ищет соответствующий контакт.
❹ Рендерит шаблон show.html.

Синтаксис извлечения значений из пути находится в первой строке кода: часть пути, которую требуется извлечь, заключается в угловые скобки `<>`, и ей присваивается имя переменной. Этот компонент пути будет извлечен и передан функции-обработчику в параметре с тем же именем.

Итак, если вы переходите к пути `/contacts/42`, значение `42` будет передано функции `contacts_view()` как значение `contact_id`.

Зная идентификатор искомого контакта, мы переходим к его поиску, для чего используется метод `find` объекта `Contact`. Затем контакт передается шаблону `show.html`, и приложение рендерит ответ.

Шаблон подробной информации о контактах

Шаблон `show.html` довольно прост. Он выводит ту же информацию, которая отображается в таблице, но в несколько ином формате (например, для печати). Если позже в приложение понадобится добавить новую функциональность (например, заметки), в шаблоне это будет удобно сделать.

И снова мы опускаем весь «хром» и обращаем внимание на содержание шаблона.

Листинг 21. Шаблон подробной информации о контактах

```
<h1>{{contact.first}} {{contact.last}}</h1>

<div>
 <div>Phone: {{contact.phone}}</div>
 <div>Email: {{contact.email}}</div>
</div>

<p>
 <a href="/contacts/{{contact.id}}/edit">Edit</a>
 <a href="/contacts">Back</a>
</p>
```

Шаблон рендерит заголовок с именем и фамилией, а под ним — дополнительную информацию о контакте и пару ссылок: для редактирования контакта и возврата к полному списку контактов.

Редактирование и удаление контакта

На следующем шаге мы займемся функциональностью ссылки редактирования (Edit). Редактирование контакта очень похоже на создание нового контакта. Как

и при добавлении нового контакта, нам понадобятся два маршрута для одного пути, но с разными методами HTTP: `GET` к `/contacts/<contact_id>/edit` возвращает форму для редактирования контакта, а `POST` к этому пути обновляет контакт.

Функциональность редактирования также будет использоваться для удаления контакта. Для этого необходимо отправить запрос `POST` к `/contacts/<contact_id>/delete`.

Рассмотрим код обработки запроса `GET`, который возвращает представление HTML для интерфейса редактирования заданного ресурса.

Листинг 22. Контроллер для редактирования контакта

```
@app.route("/contacts/<contact_id>/edit", methods=["GET"])
def contacts_edit_get(contact_id=0):
  contact = Contact.find(contact_id)
  return render_template("edit.html", contact=contact)
```

Как видите, у этого кода много общего с функциональностью вывода контактов (Show Contact). Собственно, он почти полностью совпадает с ним, если не считать шаблон: в данном случае рендерится `edit.html` вместо `show.html`.

При том что код обработчика похож на функциональность вывода контактов, шаблон `edit.html` очень похож на шаблон функциональности создания контакта (New Contact): форма в нем отправляет обновленные данные контакта по тому же URL редактирования и представляет все поля контакта в виде полей ввода для редактирования вместе с сообщениями об ошибках.

Первая часть формы:

Листинг 23. Начало формы для редактирования контакта

```
  <form action="/contacts/{{ contact.id }}/edit" method="post"> ❶
    <fieldset>
      <legend>Contact Values</legend>
       <p>
         <label for="email">Email</label>
         <input name="email" id="email" type="text"
placeholder="Email" value="{{ contact.email }}"> ❷
         <span class="error">{{ contact.errors['email'] }}</span>
       </p>
```

❶ Выдает запрос POST к пути /contacts/{{ contact.id }}/edit.

❷ Как и в случае со страницей new.html, поле ввода связывается с адресом электронной почты контакта.

Эта разметка HTML почти идентична форме `new.html`, не считая того, что эта форма будет отправлять запрос `POST` по другому пути, который определяется идентификатором обновляемого контакта. (Заметим, что вместо `POST` было бы

правильнее использовать PUT или PATCH, но эти методы недоступны в базовом HTML.)

Далее идет оставшаяся часть формы, также очень похожая на шаблон new.html, и кнопка для отправки данных формы.

Листинг 24. Тело формы для редактирования контакта

```
       <p>
         <label for="first_name">First Name</label>
         <input name="first_name" id="first_name" type="text"
placeholder="First Name"
            value="{{ contact.first }}">
         <span class="error">{{ contact.errors['first'] }}</span>
       </p>
       <p>
         <label for="last_name">Last Name</label>
         <input name="last_name" id="last_name" type="text"
placeholder="Last Name"
            value="{{ contact.last }}">
         <span class="error">{{ contact.errors['last'] }}</span>
       </p>
       <p>
         <label for="phone">Phone</label>
         <input name="phone" id="phone" type="text"
placeholder="Phone" value="{{ contact.phone }}">
         <span class="error">{{ contact.errors['phone'] }}</span>
       </p>
      <button>Save</button>
    </fieldset>
  </form>
```

В завершающей части шаблона присутствует небольшое отличие между new.html и edit.html. Под главной формой редактирования добавляется вторая форма, которая позволяет удалить контакт. Для этого она выдает запрос POST к пути /contacts/<contact_id>/delete path. По аналогии с тем, как для обновления контакта стоило бы использовать запрос PUT, для удаления контактов было бы правильнее использовать запрос HTTP DELETE. К сожалению, в базовом HTML это невозможно.

Страница завершается простой гиперссылкой на список контактов.

Листинг 25. Подвал формы для редактирования контакта

```
  <form action="/contacts/{{ contact.id }}/delete" method="post">
    <button>Delete Contact</button>
  </form>

  <p>
    <a href="/contacts/">Back</a>
  </p>
```

Если принять во внимание все сходство между шаблонами `new.html` и `edit.html`, возникает законный вопрос: почему бы не провести *рефакторинг* этих двух шаблонов для совместного использования их логики? Это правильное замечание, и, скорее всего, в реальной системе так и следовало бы поступить.

Однако мы сейчас пишем маленькое простое приложение, поэтому не будем объединять шаблоны.

ОПРЕДЕЛЕНИЕ СТРУКТУРЫ ПРИЛОЖЕНИЙ

У разработчиков, переходящих на гипермедиа-приложения после работы с JavaScript, нередко возникают проблемы с концепцией компонентов. JavaScript-ориентированные приложения часто разбиваются на мелкие компоненты на стороне клиента, которые затем собираются в единое целое. Такие компоненты часто разрабатываются и тестируются по отдельности и предоставляют удобную абстракцию для создания кода, пригодного к тестированию.

С другой стороны, структура приложений гипермедиа формируется на стороне сервера. Как было сказано выше, можно провести рефакторинг этой формы и преобразовать ее в общий шаблон для редактирования и создания. Такая реализация будет пригодна для повторного использования, и в ней будет соблюдаться принцип DRY (Don't Repeat Yourself, то есть «не повторяйтесь»).

Заметим, что структура на стороне сервера обычно менее детализирована по сравнению со стороной клиента: обычно разработчик выделяет общие *блоки*, а не множество отдельных компонентов. У такого подхода есть как свои преимущества (простота), так и недостатки (он существенно уступает компонентам на стороне клиента по уровню изоляции).

В целом в правильно сформированных приложениях гипермедиа на стороне сервера очень хорошо соблюдается принцип DRY.

Обработка запросов POST к /contacts/<contact_id>

Далее необходимо обработать запрос HTTP `POST`, отправляемый формой из шаблона `edit.html`. Мы объявим другой маршрут, который обрабатывает тот же путь, что и приведенный выше запрос `GET`.

Новый код обработчика выглядит так:

```
@app.route("/contacts/<contact_id>/edit", methods=["POST"]) ❶
def contacts_edit_post(contact_id=0):
  c = Contact.find(contact_id) ❷
  c.update(request.form['first_name'], request.form['last_name'],
request.form['phone'], request.form['email']) ❸
```

```
  if c.save(): ❹
    flash("Updated Contact!")
    return redirect("/contacts/" + str(contact_id)) ❺
  else:
    return render_template("edit.html", contact=c) ❻
```

❶ Обрабатывает запрос POST к /contacts/<contact_id>/edit.
❷ Ищет контакт по идентификатору.
❸ Обновляет контакт новой информацией от формы.
❹ Пытается сохранить контакт.
❺ В случае успеха выдается флеш-сообщение об обновлении информации контакта и происходит перенаправление на страницу с подробной информацией.
❻ В случае неудачи форма выводится заново с сообщением об ошибках.

Логика этого обработчика очень похожа на логику обработчика добавления нового контакта. Единственное реальное отличие заключается в том, что вместо создания нового экземпляра `Contact` мы ищем контакт по идентификатору, а затем вызываем для него метод `update()` со значениями, введенными в форме.

Здесь мы снова наблюдаем приятную особенность — согласованность операций CRUD, которая упрощает создание традиционных веб-приложений CRUD.

Удаление контакта

Функциональность удаления контактов эксплуатировала шаблон, использованный для редактирования контакта. Вторая форма выдает запрос HTTP `POST` к `/contacts/<contact_id>/delete`, и для этого пути также нужно будет определить обработчик.

А вот как будет выглядеть код контроллера.

Листинг 26. Контроллер для удаления контакта

```
@app.route("/contacts/<contact_id>/delete", methods=["POST"]) ❶
def contacts_delete(contact_id=0):
  contact = Contact.find(contact_id)
  contact.delete() ❷
  flash("Deleted Contact!")
  return redirect("/contacts") ❸
```

❶ Обрабатывает запрос POST к /contacts/<contact_id>/delete.
❷ Ищет контакт и вызывает для него метод delete().
❸ Создает флеш-сообщение об успешном удалении, выполняет перенаправление на страницу с основным списком контактов.

Код обработчика очень прост, потому что нам не нужна никакая логика проверки или принятия решений: мы просто ищем контакт так же, как в других

обработчиках, вызываем для него метод `delete()`, а затем выполняем перенаправление к списку контактов с флеш-сообщением об успехе.

Шаблон в данном случае не нужен, контакт удален.

Contact.app готово!

Невероятно, но это все приложение для управления контактами!

Если у вас возникли трудности с какими-то частями кода, не переживайте: мы не ожидаем, что все наши читатели будут экспертами Python или Flask (мы сами ими точно не являемся!) Чтобы воспользоваться знаниями из оставшейся части книги, от вас требуется лишь базовое понимание того, как работают язык и фреймворк.

Это маленькое и простое приложение, но и на его примере можно рассмотреть многие особенности традиционных веб-приложений 1.0: CRUD, паттерн `POST`/Redirect/`GET`, работу с логикой предметной области в контроллере, а также последовательную, ресурсно-ориентированную организацию URL.

Кроме того, это приложение является *гипермедиа-управляемым* по своей природе. Без особых усилий мы использовали REST, HATEOAS и другие концепции гипермедиа, рассмотренные выше. Мы готовы поспорить, что это простое приложение для управления контактами больше соответствует принципам REST, чем 99 % всех когда-либо созданных приложений JSON API!

Мы естественным образом получаем преимущества сетевой RESTful-архитектуры просто за счет того, что используем среду *гипермедиа* — HTML, и это хорошо. Но тогда в чем проблема? Почему не остановиться на этом и полностью не перейти на разработку веб-приложений в стиле Web 1.0?

Да, на каком-то уровне проблем нет. Старый подход к построению веб-приложений может быть полностью рабочим, особенно для таких простых приложений.

Однако наше приложение довольно «громоздкое», как и остальные приложения Web 1.0, что мы уже отмечали: каждый запрос заменяет весь экран, что создает заметный эффект мерцания при переходе между страницами. Теряется текущее состояние прокрутки. Приходится совершать больше щелчков мышью, чем в более удобных веб-приложениях.

В своем текущем виде Contact.app просто не выглядит как «современное» веб-приложение.

Не пора ли взяться за фреймворк JavaScript и JSON API, чтобы сделать приложение более интерактивным?

Нет. В этом нет необходимости.

Оказывается, приложение можно сделать более удобным и без отказа от его фундаментальной архитектуры гипермедиа.

В нескольких следующих главах мы рассмотрим `htmx`[1] — гипермедиа-ориентированную библиотеку, которая позволит нам улучшить приложение без отказа от используемого подхода гипермедиа.

Заметки об HTML: каша из фреймворков

Компоненты инкапсулируют блоки страницы вместе с их динамическим поведением. Хотя инкапсуляция поведения — эффективый механизм организации кода, она также способствует отделению элементов от окружающего контекста, что может привести к ошибочным или неадекватным отношениям между элементами. В результате возникает то, что можно назвать *кашей из компонентов*: информация скрывается в состоянии компонентов, вместо того чтобы присутствовать в разметке HTML, которая становится непонятной из-за отсутствия контекста.

Прежде чем выбирать компоненты для повторного использования, рассмотрим возможные варианты. Низкоуровневые механизмы часто генерируют более качественную разметку HTML (или по крайней мере *позволяют* ее сгенерировать). В некоторых случаях компоненты действительно *улучшают* ясность разметки HTML.

> “ Из-за того, что вы не выполняете почти никаких действий с документом HTML, потому что все необходимое будет внедряться через JavaScript, вы почти не обращаете внимания на него и на структуру страницы.
>
> *Мануэль Матузович (Manuel Matuzović),*
> *«Why I'm not the biggest fan of Single Page Applications»*[2]

Чтобы избежать появления каши из `<div>` (или каши из Markdown, или каши из компонентов), необходимо знать, какую разметку вы создаете, и иметь возможность изменить ее.

Некоторые фреймворки SPA и веб-компоненты усложняют эту задачу, создавая новые прослойки абстракции между кодом, написанным разработчиком, и генерируемой разметкой.

Хотя эти абстракции позволяют разработчикам строить более функциональные пользовательские интерфейсы или работать быстрее, их повсеместное присутствие означает, что разработчики могут обращать меньше внимания на фактическую разметку HTML (и JavaScript), отправляемую клиентам. Без должного уровня тестирования это приводит к недоступности, плохим результатам SEO и раздуванию кода.

---

[1] https://htmx.org/

[2] https://www.matuzo.at/blog/2023/single-page-applications-criticism

ЧАСТЬ II

Гипермедиа-управляемые веб-приложения в htmx

РАСШИРЕНИЕ HTML КАК ГИПЕРМЕДИА

В предыдущей главе мы создали простое гипермедиа-приложение для управления контактами в стиле Web 1.0. Оно поддерживает обычные операции CRUD для контактов, а также простой механизм поиска контактов. Мы построили его только с помощью форм и якорных тегов — традиционных элементов гипермедиа, используемых для взаимодействия с серверами. Приложение обменивается гипермедиа (HTML) с сервером через HTTP, выдавая запросы HTTP `GET` и `POST` и получая обратно полные документы HTML в ответах.

Это базовое веб-приложение, но оно определенно является гипермедиа-управляемым. Оно стабильно, в нем используются нативные технологии веб-среды, и его легко понять.

Так что же в нем не так?

К сожалению, у нашего приложения есть ряд недостатков, общих для всех приложений в стиле Web 1.0.

- С точки зрения взаимодействия с пользователем заметны обновления страницы при переходе между страницами приложения или при создании/обновлении/удалении контакта. Это объясняется тем, что каждое взаимодействие с пользователем (щелчок по ссылке или отправка данных формы) требует полного обновления страницы, а после каждого действия приходится обрабатывать совершенно новый документ HTML.
- С технической точки зрения все обновления осуществляются методом HTTP `POST`. И это несмотря на то, что существуют более логичные действия и типы запросов HTTP (такие, как `PUT` и `DELETE`) и для некоторых операций правильнее было бы использовать их. В конце концов, если необходимо удалить ресурс, разве не логичнее будет выдать для этого запрос HTTP `DELETE`? Парадоксально, но мы используем «чистую» разметку HTML, нам недоступна вся выразительная мощь HTTP, который разрабатывался специально *для* HTML.

Первый недостаток особенно хорошо заметен в приложениях в стиле Web 1.0 (таких, как наше). Именно из-за него такие приложения считаются громоздки-

ми в сравнении с усовершенствованными одностраничными (SPA) приложениями на основе JavaScript.

Проблему можно было бы решить использованием фреймворка одностраничных приложений и обновлением кода на стороне сервера, чтобы он выдавал ответы в формате JSON. Одностраничный формат избавлен от неповоротливости приложений Web 1.0, поскольку в нем веб-страница изменяется без полного обновления. В одностраничных приложениях можно изменять части модели DOM (Document Object Model) существующей страницы без необходимости заменять (и заново рендерить) всю страницу.

DOM

DOM – внутренняя модель, которую создает браузер при обработке HTML, формируя дерево «узлов» для тегов и другого контента в HTML. DOM предоставляет программный JavaScript API, позволяющий обновлять узлы страницы напрямую, без использования гипермедиа. При помощи этого API код JavaScript может вставлять новый контент, а также удалять или обновлять существующий – и все это в обход обычного механизма запросов браузера.

У SPA есть несколько разновидностей, но, как обсуждалось в главе 1, в самом распространенном в наши дни решении DOM привязывается к модели JavaScript, после чего фреймворк SPA (такой, как React[1] или Vue[2]) обеспечивает *реактивное* обновление DOM при обновлении модели JavaScript: вы вносите изменения в объект JavaScript, хранимый локально в памяти браузера, а веб-страница «волшебным образом» обновляет свое состояние в соответствии с изменениями в модели.

В приложениях этого типа обмен данными с сервером обычно осуществляется через JSON Data API; приложение жертвует преимуществами гипермедиа, чтобы обеспечить более плавное и удобное взаимодействие с пользователем.

Многие веб-разработчики даже не рассматривают подход гипермедиа из-за бытующего мнения о том, что приложения в стиле Web 1.0 «устарели».

Вторая, техническая, проблема, уже упомянутая ранее, может показаться формальной. И мы первыми согласимся с тем, что разговоры о REST и о том, какое действие HTTP следует использовать для конкретной операции, быстро утомляют. И все же очень странно, что при использовании базового HTML невозможно использовать всю функциональность HTTP!

Ведь такого быть не должно, не так ли?

---

[1] https://reactjs.org/

[2] https://vuejs.org/

Гиперссылки крупным планом

Как выясняется, интерактивность нашего приложения можно существенно улучшить и решить обе проблемы, не прибегая к SPA. Для этого можно воспользоваться *гипермедиа-ориентированной* библиотекой JavaScript — `htmx`[1]. Авторы книги разработали `htmx` специально для того, чтобы расширить HTML как среду гипермедиа и устранить перечисленные выше недостатки устаревших приложений HTML (а также ряд других недостатков).

Прежде чем разбираться в том, как `htmx` помогает улучшить UX (user experience, пользовательский опыт) приложений в стиле Web 1.0, вернемся к тегу гиперссылки/якоря из главы 1. Напомним, что гиперссылка является так называемым *гипермедиа-элементом управления* — механизмом, который описывает определенное взаимодействие с сервером посредством прямого и полного кодирования информации об этом взаимодействии в самом элементе.

Возьмем простой якорный тег, который при интерпретации браузером создает гиперссылку на веб-сайт этой книги.

Листинг 27. Простая гиперссылка

```
<a href="https://hypermedia.systems/">
 Системы гипермедиа
</a>
```

Рассмотрим подробно, что же происходит при отображении этой ссылки.

- Браузер выводит на экран текст «Системы гипермедиа» — вероятно, с визуальными признаками, указывающими на то, что по нему можно щелкнуть.
- Затем пользователь щелкает на тексте...
- ...и браузер выдает запрос HTTP `GET` к https://hypermedia.systems...
- Браузер загружает в окне тело HTML ответа HTTP, заменяя текущий документ.

Итак, для простой гипермедиа-ссылки характерны четыре особенности, причем три последние образуют механизм, отличающий гиперссылку от «обычного» текста, а следовательно, превращающий ее в элемент гипермедиа.

А теперь подумаем, как *обобщить* эти три последние особенности гиперссылок.

Почему только якорные теги и формы?

Почему же выделяются якорные теги (и формы)?

Почему другие элементы не могут выдавать запросы HTTP?

[1] https://htmx.org/

Например, почему элементы `button` не могут выдавать запросы HTTP? Казалось бы, естественно упаковать `button` в тег формы, просто чтобы реализовать функциональность удаления контактов в приложении.

Вероятно, другие элементы тоже могли бы выдавать запросы HTTP. Может быть, другие элементы тоже могли бы действовать как гипермедиа сами по себе.

Это первая возможность обобщения HTML в виде гипермедиа.

ПЕРВАЯ ВОЗМОЖНОСТЬ

HTML можно расширить, чтобы *любой* элемент мог выдавать запрос к серверу и работать как гипермедиа.

Почему только события щелчков и отправки данных?

Рассмотрим событие, которое инициирует для ссылки запрос к серверу: событие щелчка.

Что же такого особенного в щелчках (в случае якорных тегов) или отправке данных (в случае форм)? Ведь это всего лишь два из многих-многих событий DOM. Нажатие кнопки мыши, отпускание клавиши, потеря фокуса — все эти события тоже могут использоваться для выдачи запросов HTTP.

Почему бы не наделить все эти события возможностью инициировать запросы?

Так мы получаем вторую возможность расширения выразительности HTML.

ВТОРАЯ ВОЗМОЖНОСТЬ

HTML можно расширить так, чтобы *любое* событие — не только щелчок, как в случае гиперссылок, — могло инициировать запросы HTTP.

Почему только GET и POST?

Переходя на технический уровень обсуждения, мы возвращаемся к уже упомянутой проблеме: базовый HTP предоставляет доступ только к действиям `GET` и `POST` протокола HTTP.

Сокращение «HTTP» означает «протокол передачи гипертекста» (Hypertext Transfer Protocol), однако формат, для которого он проектировался, — HTML — поддерживает только два из пяти типов запросов, предназначенных для разработчиков. Чтобы получить доступ к трем оставшимся: `DELETE`, `PUT` и `PATCH`, *придется* использовать JavaScript и выдавать запросы AJAX.

Вспомним, что должны были представлять разные типы запросов HTTP, по замыслу проектировщиков.

- `GET` соответствует «получению» представления ресурса от URL: это чистое чтение без изменения ресурса.
- `POST` отправляет сущность (или данные) заданному ресурсу. Часто запрос приводит к созданию или изменению ресурса и изменению состояния.
- `PUT` отправляет сущность (или данные) заданному ресурсу для обновления или замены. Также часто приводит к изменению состояния.
- `PATCH` аналогичен `PUT`, но подразумевает частичное обновление и изменение состояния вместо полной замены сущности.
- `DELETE` удаляет заданный ресурс.

Эти операции похожи на операции CRUD, рассмотренные в главе 2. Предоставляя доступ только к двум из пяти операций, HTML ограничивает возможность использования всего потенциала HTTP.

Так мы приходим к третьей возможности расширения выразительности HTML.

ТРЕТЬЯ ВОЗМОЖНОСТЬ

! HTML можно расширить так, чтобы он предоставлял доступ к трем недостающим методам HTTP: PUT, PATCH и DELETE.

Почему заменяется вся страница целиком?

Наконец, последняя особенность гиперссылки: по щелчку на ней заменяется вся страница целиком.

Оказывается, эта техническая деталь и есть главный виновник плохого взаимодействия с пользователем в приложениях Web 1.0. Полное обновление страницы может вызвать мерцание неоформленного контента, когда контент резко появляется на экране в состоянии перехода от исходной формы к итоговому оформленному виду. Кроме того, при обновлении теряется положение прокрутки для пользователя (прокрутка возвращается к началу страницы), теряется фокус ввода на элементе и т. д.

Но если задуматься, ничто не обязывает заменять весь документ при обмене гипермедиа. Так мы приходим к четвертой, последней и, пожалуй, самой важной возможности обобщения HTML.

ЧЕТВЕРТАЯ ВОЗМОЖНОСТЬ

! HTML можно расширить так, чтобы ответы на запросы заменяли элементы *внутри* текущего документа (вместо обязательной замены *всего* документа).

На самом деле эта концепция далеко не нова. Тед Нельсон в своей книге «Literary Machines» 1980 года ввел термин *виртуальное включение* (transclusion) для представления идеи включения контента в существующий документ по гипермедиа-ссылке. Если бы в HTML поддерживался такой стиль «динамического виртуального включения», то гипермедиа-управляемые приложения были бы больше похожи на одностраничные приложения, в которых пользовательское действие или сетевой запрос обновляют только отдельную часть DOM.

Расширение HTML как гипермедиа с htmx

Эти четыре возможности показывают, как расширить текущую функциональность HTML *полностью в рамках* гипермедиа-модели веб-среды. В фундаментальных свойствах HTML, HTTP, браузера и т. д. никаких радикальных изменений не будет. Скорее эти способы развития *существующей функциональности* HTML просто позволят добиться *большего* с HTML.

`Htmx` — библиотека JavaScript, расширяющая HTML именно таким образом. Ей будут посвящены несколько следующих глав этой книги. Еще раз подчеркнем, что `htmx` — не единственная гипермедиа-ориентированная библиотека JavaScript (другие прекрасные примеры — Unpoly[1] и Hotwire[2]), но `htmx` предлагает расширение HTML как гипермедиа в самом чистом виде.

Установка и использование htmx

С практической точки зрения начинающего пользователя, `htmx` — простая, свободная от зависимостей и автономная библиотека JavaScript, которую можно включить в веб-приложение всего лишь добавлением тега `script` в элемент `head`.

Благодаря этой простоте при установке можно воспользоваться такими средствами, как общедоступные сети поставки контента, или CDN (Content Delivery Network).

В следующем примере популярная CDN `unpkg`[3] применяется для установки версии 1.9.2 этой библиотеки. Для проверки того, что доставленный контент JavaScript совпадает с ожидаемым, используется хеш целостности данных. Значение SHA можно найти на сайте `htmx`.

Скрипт также помечается `crossorigin="anonymous"`, чтобы в CDN не передавались идентификационные данные.

---

[1] https://unpoly.com/

[2] https://hotwire.dev/

[3] https://unpkg.com/

Листинг 28. Установка htmx

```
<head>
<script src="https://unpkg.com/htmx.org@1.9.2"
    integrity="sha384-
L6OqL9pRWyyFU3+/bjdSri+iIphTN/bvYyM37tICVyOJkWZLpP2vGn6VUEXgzg6h"
    crossorigin="anonymous"></script>

</head>
```

Если вы привыкли к современной разработке на JavaScript с ее сложными системами сборки и большим количеством зависимостей, такая простота установки `htmx` станет для вас приятным сюрпризом. Все в духе ранних веб-приложений, когда было достаточно добавить тег `script` и оно «просто работало».

Если вы не хотите использовать CDN, просто загрузите `htmx` в свою локальную систему и измените тег `script`, чтобы он указывал на место хранения статических ресурсов. А может, у вас имеется система сборки, которая устанавливает зависимости автоматически? В таком случае можно воспользоваться именем `npm` (Node Package Manager) для библиотеки: `htmx.org` и установить ее обычным способом, который поддерживается вашей системой сборки.

После того как библиотека `htmx` установлена, ее можно начинать использовать.

JavaScript не требуется...

И здесь мы подходим к самому интересному: `htmx` не требует от своих пользователей писать код JavaScript.

Вместо этого для реализации более динамического поведения можно использовать *атрибуты*, помещаемые непосредственно в элементы HTML. `Htmx` расширяет HTML как гипермедиа, причем библиотека спроектирована так, чтобы это расширение воспринималось как абсолютно естественное и соотносящееся с существующими концепциями HTML. Подобно тому как в якорном теге есть атрибут `href` для определения извлекаемого URL, а в форме — атрибут `action` для определения URL отправки формы, `htmx` использует *атрибуты* HTML для определения URL, по которому должен выдаваться запрос HTTP.

Выдача запросов HTTP

Начнем с первой особенности `htmx`: возможности выдачи запросов HTML любым элементом веб-страницы. Это основная функциональность `htmx`, и она представлена пятью атрибутами, которые могут использоваться для выдачи пяти разных типов запросов HTTP, ориентированных на разработчика:

- `hx-get` — выдает запрос HTTP `GET`;
- `hx-post` — выдает запрос HTTP `POST`;
- `hx-put` — выдает запрос HTTP `PUT`;
- `hx-patch` — выдает запрос HTTP `PATCH`;
- `hx-delete` — выдает запрос HTTP `DELETE`.

Каждый из этих атрибутов при включении в элемент сообщает библиотеке `htmx`: «Когда пользователь щелкает на этом элементе (или иным образом взаимодействует с ним), выдать запрос HTTP указанного типа».

Значения атрибутов аналогичны значениям `href` для якорных тегов и `action` для форм: вы задаете URL, по которому должен выдаваться запрос HTTP указанного типа. Как правило, это делается указанием пути относительно сервера.

Например, если вы хотите, чтобы кнопка выдавала запрос `GET` к `/contacts`, используйте следующую разметку HTML.

Листинг 29. Простая кнопка на основе htmx

```
<button hx-get="/contacts"> ❶
 Get Contacts
</button>
```

❶ Простая кнопка, которая выдает запрос HTTP GET к /contacts.

Библиотека `htmx` обнаруживает у кнопки атрибут `hx-get` и подключает логику JavaScript для выдачи AJAX-запроса HTTP `GET` к пути `/contacts`, когда пользователь щелкает на этой кнопке.

Все просто и соответствует логике остального кода HTML.

Всего лишь HTML

И тут мы приходим, пожалуй, к самому важному, что необходимо понять о библиотеке `htmx`: она ожидает, что ответ на запрос AJAX *будет закодирован в HTML*. `Htmx` является расширением HTML. Встроенный гипермедиа-элемент управления, такой как якорный тег, обычно получает на созданный им запрос HTTP ответ HTML. Точно так же `htmx` ожидает, что сервер будет отвечать на выдаваемые запросы разметкой HTML.

Это может быть неожиданностью для веб-разработчиков, привыкших отвечать на запросы AJAX в формате JSON — безусловно, самом распространенном формате ответа для таких запросов. Однако запросы AJAX — всего лишь запросы HTTP, и не существует правил, требующих использовать в них JSON. Вспомните, что сокращение AJAX означает Asynchronous JavaScript & XML, то есть

«асинхронный JavaScript и XML», так что JSON в действительности отходит от формата, который изначально планировался для этого API: XML.

`Htmx` просто идет в другом направлении и ожидает получить HTML.

Htmx и «обычные» ответы HTML

Между ответами HTTP на «обычные» запросы HTTP от якорных тегов/форм и ответами на основе `htmx` существует важное отличие: в случае запросов, инициируемых `htmx`, ответы могут содержать *неполные* фрагменты HTML.

Как вы вскоре увидите, во взаимодействиях на основе `htmx` замены всего документа обычно не происходит. Вместо этого осуществляется «виртуальное включение» контента *внутрь* существующего документа. Из-за этого зачастую не обязательно или даже нежелательно передавать весь документ HTML от сервера к браузеру.

Эта особенность может использоваться для экономии пропускной способности канала связи, а также для ускорения загрузки ресурсов. От сервера к клиенту передается меньший объем контента, пропадает необходимость повторной обработки тега `head` с таблицами стилей, тегами `script` и т. д.

При щелчке на кнопке Get Contacts (Контакты) *частичный* ответ HTML может выглядеть примерно так:

Листинг 30. Частичный ответ HTML на запрос htmx

```
<ul>
 <li><a href="mailto:joe@example.com">Joe</a></li>
 <li><a href="mailto:sarah@example.com">Sarah</a></li>
 <li><a href="mailto:fred@example.com">Fred</a></li>
</ul>
```

Это просто неупорядоченный список контактов с несколькими элементами, на которые можно щелкнуть. Обратите внимание: здесь нет открывающего тега `html`, открывающего тега `head` и т. д.; перед вами простой список HTML без дополнительных элементов. Ответ в реальном приложении может содержать более сложную разметку HTML, но и такая разметка не обязательно должна содержать полную страницу HTML: это может быть «внутренний» контент представления HTML для этого ресурса.

Ответ с простым списком идеально подходит для `htmx`. `Htmx` просто берет возвращенный контент и подставляет его в DOM на место некоторого элемента страницы. (О том, куда именно он будет помещен в DOM, мы расскажем чуть позже). Такая подстановка HTML выполняется быстро и эффективно, потому что она использует существующий встроенный парсер HTML браузера, не требуя выполнения значительного объема JavaScript на стороне клиента.

Этот компактный ответ HTML показывает, как htmx остается в парадигме гипермедиа: как и «обычный» гипермедиа-элемент управления в «обычном» веб-приложении, гипермедиа передается клиенту без состояния и по схожей схеме. Кнопка лишь чуть более усложняет построение веб-приложения с использованием гипермедиа.

Другие элементы как цели

Итак, htmx выдает запрос и получает разметку HTML в ответе. Полученный контент должен быть подставлен в существующую страницу (вместо замены всей страницы). Куда же помещать этот новый контент?

Как выясняется, по умолчанию htmx просто помещает возвращенный контент внутрь элемента, инициировавшего запрос. Для кнопки такой вариант *не* подходит: мы получим список контактов, грубо вставленный в элемент кнопки. Это выглядит нелепо и с очевидностью не то, что нам нужно.

К счастью, htmx предоставляет другой атрибут — hx-target, при помощи которого можно точно указать, *где* именно в DOM должен быть размещен новый контент. Значение атрибута hx-target представляет собой *селектор* CSS (Cascading Style Sheet); он позволяет задать элемент, в который будет вставлен новый контент гипермедиа.

Добавим тег div, включающий кнопку с идентификатором main. Затем назначим тег div целевым для ответа.

Листинг 31. Простая кнопка на основе htmx

```
<div id="main"> ❶

 <button hx-get="/contacts" hx-target="#main"> ❷
  Get Contacts
 </button>

</div>
```

❶ Элемент div, заключающий в себе кнопку.
❷ Атрибут hx-target задает цель ответа.

К кнопке добавлен атрибут hx-target="#main", где #main — селектор CSS, означающий «объект с идентификатором main».

Использование селекторов CSS означает, что htmx строится на основе знакомых стандартных концепций HTML. Это позволяет минимизировать количество концепций для работы с htmx.

Как будет выглядеть разметка HTML на стороне клиента после того, как пользователь щелкнет на кнопке, а ответ будет получен и обработан?

Она будет выглядеть примерно так:

Листинг 32. HTML после завершения запроса htmx

```
<div id="main">
 <ul>
  <li><a href="mailto:joe@example.com">Joe</a></li>
  <li><a href="mailto:sarah@example.com">Sarah</a></li>
  <li><a href="mailto:fred@example.com">Fred</a></li>
 </ul>
</div>
```

Контент HTML из ответа подставлен в `div`, заменяя кнопку, инициировавшую запрос. Виртуальное включение! И все это происходит «на заднем плане» через механизм AJAX без грубого обновления страницы.

Стили подстановки

Предположим, вы не хотите загружать контент из ответа сервера *внутрь* `div` как дочерние элементы. Возможно, по какой-то причине вы предпочитаете *заменить* весь элемент `div` ответом. Для таких случаев `htmx` предоставляет другой атрибут, `hx-swap`, который позволяет точно указать, *как именно* контент должен быть подставлен в DOM.

Атрибут `hx-swap` поддерживает следующие значения:

- `innerHTML` — используется по умолчанию, заменяет внутреннюю разметку HTML целевого элемента;
- `outerHTML` — заменяет весь целевой элемент ответом;
- `beforebegin` — вставляет ответ перед целевым элементом;
- `afterbegin` — вставляет ответ перед первым потомком целевого элемента;
- `beforeend` — вставляет ответ после последнего потомка целевого элемента;
- `afterend` — вставляет ответ после целевого элемента;
- `delete` — удаляет целевой элемент независимо от ответа;
- `none` — подстановка не выполняется.

Первые два значения, `innerHTML` и `outerHTML`, заимствованы из стандартных свойств DOM, позволяющих заменить контент внутри элемента или весь элемент соответственно.

Следующие четыре значения заимствованы из DOM API `Element.insertAdjacentHTML()`, позволяющего поместить элемент или элементы вокруг заданного элемента разными способами.

Последние два значения, `delete` и `none`, специфичны для `htmx`. Первое удаляет целевой элемент из DOM, тогда как второе не делает ничего (например, если вы хотите работать только с заголовками ответов — продвинутая техника, которую мы рассмотрим далее в книге).

И снова мы видим, что `htmx` максимально приближен к существующим вебстандартам, сводя к минимуму концептуальную нагрузку, необходимую для использования библиотеки.

Рассмотрим случай, в котором вместо замены `innerHTML` контента основного тега `div`, приведенного выше, мы будем заменять весь `div` ответом HTML.

Для этого достаточно внести в кнопку небольшое изменение — добавить новый атрибут `hx-swap`.

Листинг 33. Замена всего элемента div

```
<div id="main">

 <button hx-get="/contacts" hx-target="#main" hx-swap="outerHTML"> ❶
  Get The Contacts
 </button>

</div>
```

❶ Атрибут hx-swap указывает, как должен быть подставлен новый контент.

Теперь при получении ответа весь элемент `div` будет заменен контентом гипермедиа.

Листинг 34. Разметка HTML после завершения запроса htmx

```
<ul>
 <li><a href="mailto:joe@example.com">Joe</a></li>
 <li><a href="mailto:sarah@example.com">Sarah</a></li>
 <li><a href="mailto:fred@example.com">Fred</a></li>
</ul>
```

Как видите, с таким изменением целевой элемент `div` был полностью исключен из DOM, а список, возвращенный в виде ответа, заменил его.

Далее в книге будут приведены другие примеры использования `hx-swap`, например для реализации бесконечной прокрутки в приложении для управления контактами.

Заметим, что с атрибутами `hx-get`, `hx-post`, `hx-put`, `hx-patch` и `hx-delete` мы реализовали две из четырех возможностей улучшения базового HTML, названных выше.

- Возможность 1: теперь запросы HTTP могут выдаваться *любыми* элементами (в данном случае используется кнопка).

- Возможность 3: теперь можно выдавать *любые* запросы HTTP по своему усмотрению, в частности `PUT`, `PATCH` и `DELETE`.

А с помощью `hx-target` и `hx-swap` мы избавились от третьего недостатка: требования о замене всей страницы.

- Возможность 4: теперь можно заменить любой элемент страницы по своему усмотрению посредством виртуального включения, и это можно сделать любым предпочитаемым способом.

Таким образом, имея семь относительно простых дополнительных атрибутов, нам удалось избавиться от большинства недостатков HTML как среды гипермедиа, перечисленных выше.

Что дальше? Вспомните, о какой еще особенности мы упоминали: тот факт, что только событие `click` (для якорного тега) или `submit` (для формы) может инициировать запрос HTTP. Посмотрим, как преодолеть это ограничение.

Использование событий

До сих пор мы использовали кнопку для выдачи запроса средствами `htmx`. Вероятно, вы интуитивно поняли, что кнопка выдаст запрос при щелчке по ней, потому что кнопки именно для этого и нужны: чтобы на них щелкали.

Да, по умолчанию при закреплении за кнопкой `hx-get` или другой аннотации управления запросами этот запрос будет выдан по щелчку на кнопке.

Однако `htmx` обобщает эту концепцию события, инициирующего запрос, в форме — совершенно верно — еще одного атрибута: `hx-trigger`. Атрибут `hx-trigger` позволяет задать одно или несколько событий, при наступлении которых элемент будет выдавать запрос HTTP.

Часто использовать `hx-trigger` не придется, потому что инициирующее событие по умолчанию и будет именно тем, что вам нужно. Инициирующее событие по умолчанию зависит от типа элемента, и оно должно быть интуитивно понятно.

- Запросы элементов `input`, `textarea` и `select` инициируются событием `change`.
- Запросы элементов `form` инициируются событием `submit`.
- Запросы всех остальных элементов инициируются событием `click`.

Чтобы продемонстрировать, как работает `hx-trigger`, представим следующую ситуацию: запрос для кнопки должен инициироваться, когда на нее наведен указатель мыши. Конечно, это нельзя назвать *хорошим* паттерном UX, но будьте снисходительны: мы взяли эту ситуацию только для примера.

Чтобы отреагировать на наведение указателя мыши в границах кнопки, добавим к кнопке следующий атрибут.

Листинг 35. Кнопка, инициирующая запрос по наведению указателя мыши

```
<div id="main">

 <button hx-get="/contacts" hx-target="#main" hx-swap="outerHTML" hx-
trigger="mouseenter"> ❶
  Get Contacts
 </button>

</div>
```

❶ Выдает запрос по событию mouseenter.

После того как вы определите атрибут `hx-trigger`, запрос будет инициироваться каждый раз, когда указатель мыши попадает в границы кнопки. Просто, но эффективно.

Попробуем кое-что более реалистичное и потенциально полезное: добавим поддержку комбинации клавиш для загрузки контактов, Ctrl+L (от слова «Load» — загрузка). Для этого понадобится дополнительный синтаксис, поддерживаемый атрибутом `hx-trigger`: фильтры событий и дополнительные аргументы.

Фильтры событий — механизм для определения того, должно ли некоторое событие инициировать запрос. Чтобы добавить фильтр к событию, следует указать его в квадратных скобках после события: `someEvent[someFilter]`. Сам фильтр представляет собой выражение JavaScript, которое будет оцениваться при возникновении заданного события. Если результат окажется квазиистинным (в смысле JavaScript), то запрос инициируется, а если нет — не инициируется.

В случае комбинаций клавиш необходимо перехватывать событие `keyup` в дополнение к событию `click`.

Листинг 36. Добавление инициирующего события keyup в начале

```
<div id="main">

 <button hx-get="/contacts" hx-target="#main" hx-swap="outerHTML" hx-
trigger="click, keyup"> ❶
  Get The Contacts
 </button>

</div>
```

❶ Инициируется по двум событиям.

Обратите внимание на разделенный запятыми список событий, которые инициируют срабатывание элемента. Это позволяет реагировать на несколько по-

тенциальных инициирующих событий. Нам все еще требуется реагирование на событие `click` для загрузки контактов в дополнение к обработке комбинации клавиш Ctrl+L.

К сожалению, у добавления `keyup` есть две проблемы. Запросы теперь инициируются по *любому* событию `keyup`. Еще хуже то, что они инициируются только при наступлении события `keyup` *в границах* кнопки. Пользователю придется перейти к кнопке клавишей Tab, чтобы сделать ее активной, а затем нажать комбинацию клавиш.

Исправим эти два недостатка. Для первого воспользуемся фильтром, который проверяет, что клавиши Ctrl и L нажаты одновременно.

Листинг 37. Применение фильтра для улучшения обработки keyup

```
<div id="main">

 <button hx-get="/contacts" hx-target="#main" hx-swap="outerHTML" hx-
trigger="click, keyup[ctrlKey && key == 'l']"> ❶
  Get Contacts
 </button>

</div>
```

❶ Событию keyup назначается фильтр, поэтому клавиши Ctrl и L должны быть нажаты одновременно.

Фильтром срабатывания в данном случае является выражение `ctrlKey && key == 'l'`. Его можно прочитать так: «Событие отпускания клавиши, у которого свойство `ctrlKey` равно `true`, а свойство `key` равно `l`». Обратите внимание: свойства `ctrlKey` и `key` разрешаются событийно, а не по глобальному пространству имен, что позволяет легко фильтровать свойства событий. В качестве фильтра может использоваться любое выражение: например, ничто не мешает вызвать глобальную функцию JavaScript.

Итак, этот фильтр ограничивает события `keyup` и инициирует запрос только при нажатии Ctrl+L. Однако остается вторая проблема: в текущем состоянии запрос будет инициирован только событиями `keyup` *внутри* кнопки.

Если вдруг вы не знакомы с моделью всплывающих событий JavaScript: события обычно «всплывают» к родительским элементам. Таким образом, такое событие, как `keyup`, будет сначала инициироваться для элемента, обладающего фокусом, затем для его родительского (внешнего) элемента, и так далее, пока не будет достигнут объект `document` верхнего уровня, являющийся корневым для всех остальных элементов.

Для поддержки глобальных комбинаций клавиш, которые работают независимо от того, какой элемент обладает фокусом, мы воспользуемся механизмом всплывающих событий и возможностью, которую предоставляет атрибут

hx-trigger: прослушиванием событий *других элементов*. Синтаксис прослушивания основан на модификаторе from:, который добавляется после имени события и позволяет указать конкретный элемент для прослушивания заданного события при помощи селектора CSS.

В данном случае для прослушивания должен использоваться элемент body, который является родительским для всех видимых элементов страницы.

Обновленный атрибут hx-trigger выглядит так:

Листинг 38. Улучшенная версия: прослушивание keyup для body

```
<div id="main">

 <button hx-get="/contacts" hx-target="#main" hx-swap="outerHTML" hx-
trigger="click, keyup[ctrlKey && key == 'L'] from:body"> ❶
  Get The Contacts
 </button>

</div>
```

❶ Прослушивает событие keyup для тега body.

Теперь, помимо щелчков на ней, кнопка будет прослушивать события keyup для тела страницы. Таким образом, она будет выдавать запрос, когда по ней делается щелчок и когда кто-то нажимает Ctrl+L в пределах тела страницы.

Итак, мы задали удобную комбинацию клавиш для своего гипермедиа-управляемого приложения.

Атрибут hx-trigger поддерживает много других модификаторов, и он сложнее других атрибутов htmx. Дело в том, что механизм событий в целом нетривиален, и чтобы он правильно работал, необходимо учитывать множество нюансов. Впрочем, зачастую события по умолчанию оказывается достаточно, и обычно при использовании htmx обращаться к дополнительной функциональности hx-trigger не придется.

Даже с более сложными спецификациями событий, как в только что добавленном случае с комбинацией клавиш, стиль htmx больше напоминает *декларативный*, а не *императивный*. В результате приложения на основе htmx «воспринимаются» как стандартные приложения Web 1.0, в отличие от приложений с большим объемом кода JavaScript.

Htmx как расширенный HTML

Смотрите-ка! С hx-trigger мы реализовали последнюю из возможностей улучшения HTML, перечисленных в начале главы.

- Возможность 2: запросы HTTP может инициировать *любое* событие.

Итого получаем восемь — пересчитайте, если не верите, — *восемь* атрибутов, которые полностью укладываются в ту же концептуальную модель, что и обычный HTML, а расширение HTML как гипермедиа открывает целый мир новых перспектив для пользовательских взаимодействий.

Ниже собраны все возможности улучшения HTML и атрибуты `htmx`, которые при этом используются.

Любой элемент должен быть способен выдавать запросы HTTP.

`hx-get`, `hx-post`, `hx-put`, `hx-patch`, `hx-delete`

Любое событие должно иметь возможность инициировать запрос HTTP.

`hx-trigger`

Должны быть доступны все действия HTTP.

`hx-put`, `hx-patch`, `hx-delete`

Запросы должны иметь возможность заменять любую часть страницы (виртуальное включение).

`hx-target`, `hx-swap`

Передача параметров запроса

До сих пор мы рассматривали только ситуацию, в которой кнопка выдает простой запрос `GET`. На концептуальном уровне она очень близка к тому, что может делать якорный тег. Однако в приложениях на основе HTML существует другой встроенный гипермедиа-элемент управления — формы. Формы используются для передачи серверу в запросе дополнительной информации, помимо URL.

Эта информация сохраняется в полях ввода и других элементах формы при помощи разных видов тегов ввода, доступных в HTML.

`Htmx` позволяет включать такую дополнительную информацию способом, который моделирует функциональность самого HTML.

Внешние формы

Простейший способ передачи входящих значений с запросом в `htmx` — включение элемента, выдающего запрос, в тег `form`.

Возьмем исходную кнопку для загрузки контактов и переделаем ее для поиска контактов.

Листинг 39. Кнопка поиска на основе htmx

```
<div id="main">
 <form> ❶
   <label for="search">Search Contacts:</label>
   <input id="search" name="q" type="search" placeholder="Search
Contacts"> ❷
   <button hx-post="/contacts" hx-target="#main"> ❸
    Search Contacts
   </button>
 </form>

</div>
```

❶ Во внешнем теге формы будут отправлены все значения из полей ввода.
❷ Новое поле для ввода текста поискового запроса.
❸ Кнопка преобразована для hx-post.

Здесь мы добавили тег `form`, включающий саму кнопку и поле поиска, которое может использоваться для ввода условий поиска.

Теперь, когда пользователь щелкнет на кнопке, значение поля ввода с идентификатором `search` будет включено в запрос. Это происходит благодаря присутствию тега `form`, включающего как кнопку, так и поле ввода: при инициировании запроса на основе `htmx` библиотека `htmx` ищет в иерархии DOM внешнюю форму, и если она обнаружена, включает все значения из этой формы (иногда это называется сериализацией формы).

Возможно, вы заметили, что кнопка переключилась с запроса `GET` на запрос `POST`. Дело в том, что по умолчанию `htmx` *не включает* ближайшую внешнюю форму для запросов `GET`, но *включает* ее для всех остальных типов запросов.

На первый взгляд такой подход может показаться странным, но он позволяет избежать засорения URL, используемых в формах, лишней информацией при работе с элементами истории просмотра (об этом чуть позже). Кроме того, всегда можно включить значения внешней формы в элемент, использующий `GET`, при помощи атрибута `hx-include`, рассматриваемого в следующем разделе.

Включение входящих значений

Хотя самый популярный способ работы с полями ввода в запросах `htmx` — включать все входящие значения, которые вы считаете нужным включить, это не всегда возможно или желательно: у тегов форм могут быть свои особенности, когда их нельзя размещать в некоторых местах документа HTML. Хороший пример — элементы строк таблицы (`tr`): тег `form` не является допустимым до-

черним элементом или родителем строк таблицы, так что форму не удастся разместить внутри или вокруг строки данных в таблице.

Для таких случаев `htmx` предоставляет механизм включения входящих значений в запросы: атрибут `hx-include`. Атрибут `hx-include` позволяет выбрать входящие значения, которые требуется включить в запрос, при помощи селекторов CSS.

Ниже показан предыдущий пример, переработанный для включения значений ввода без использования формы.

Листинг 40. Кнопка поиска на основе htmx с атрибутом hx-include

```
<div id="main">

 <label for="search">Search Contacts:</label>
 <input id="search" name="q" type="search" placeholder="Search Contacts">
 <button hx-post="/contacts" hx-target="#main" hx-include="#search"> ❶
  Search Contacts
 </button>

</div>
```

❶ hx-include используется для включения значений непосредственно в запрос.

Атрибут `hx-include` получает селектор CSS и позволяет указать, какие значения должны быть отправлены с запросом. Это может быть удобно, если возникают затруднения при сопоставлении элемента, выдающего запрос, со всеми нужными полями ввода.

Также эта возможность может пригодиться, если требуется отправить значения с запросом `GET` и переопределить поведение `htmx` по умолчанию.

Относительные селекторы CSS

Атрибут `hx-include` (а на самом деле большинство атрибутов, получающих селекторы CSS) также поддерживает *относительные* селекторы CSS. Он позволяет задать селектор CSS *относительно* того элемента, в котором объявляется. Несколько примеров:

- `closest` — найти ближайший родительский элемент, соответствующий заданному селектору (например, `closest form`);
- `next` — найти следующий элемент (с прямым сканированием), соответствующий заданному селектору (например, `next input`);
- `previous` — найти предыдущий элемент (с обратным сканированием), соответствующий заданному селектору (например, `previous input`);

- `find` — найти внутри текущего элемента следующий элемент, соответствующий заданному селектору (например, `find input`);
- `this` — текущий элемент.

Использование относительных селекторов CSS часто позволяет избежать генерирования идентификаторов для элементов, так как вместо этого можно воспользоваться знанием локальной структуры элементов.

Встроенные значения

Последний способ включения значений в запросы, управляемые `htmx`, — использование атрибута `hx-vals`, который позволяет добавить к запросу «статические» значения. Это может быть полезно, если в запрос необходимо включить дополнительную информацию, но не внедрять ее, например, в скрытые поля (стандартный механизм для добавления скрытой информации в HTML).

Пример использования `hx-vals`:

Листинг 41. Кнопка на основе htmx с атрибутом hx-vals

```
<button hx-get="/contacts" hx-vals='{"state":"MT"}'> ❶
 Get The Contacts In Montana
</button>
```

❶ hx-vals — значение JSON, включаемое в запрос.

Параметр `state` со значением `MT` включается в запрос `GET`, что приводит к формированию пути с параметрами вида `/contacts?state=MT`. Обратите внимание, что значение атрибута `hx-vals` заключено в одинарные кавычки. Дело в том, что JSON требует использования двойных кавычек в обязательном порядке; следовательно, чтобы избежать экранирования, необходимо использовать для значения атрибута форму с одинарными кавычками.

В значение `hx-vals` можно включить префикс `js:` и передать значения, вычисляемые на момент запроса, что может быть полезно для включения таких данных, как динамические переменные или значение из сторонней библиотеки JavaScript.

Например, если переменная `state` вычисляется динамически каким-нибудь кодом JavaScript и существует функция JavaScript `getCurrentState()`, которая возвращает текущий выбранный штат, ее можно включить динамически в запросы `htmx`.

Листинг 42. Динамическое значение

```
<button hx-get="/contacts" hx-vals='js:{"state":getCurrentState()}'> ❶
 Get The Contacts In The Selected State
</button>
```

❶ С префиксом js: это выражение вычисляется в момент отправки данных.

Эти три механизма, использующие теги `form`, атрибут `hx-include` и атрибут `hx-vals`, позволяют включать значения в запросы гипермедиа `htmx` привычным для HTML способом, при этом обеспечивая гибкость, необходимую для достижения цели.

Поддержка истории

Чтобы завершить наш обзор `htmx`, осталось рассмотреть последний аспект ее функциональности: поддержку истории браузера. При использовании обычных ссылок и форм HTML браузер отслеживает все посещенные страницы. Кнопка Назад (Back) служит для возврата к предыдущей странице, а кнопка Вперед (Forward) — для перехода к исходной странице, на которой вы находились.

Концепция истории просмотра была одним из уникальных достижений на заре веб-технологий. К сожалению, выясняется, что переход на парадигму приложений SPA усложняет работу с историей. Запрос AJAX сам по себе не регистрирует веб-страницу в истории браузера, и это правильно: запрос AJAX может не иметь отношения к состоянию веб-страницы (возможно, он просто регистрирует некую активность в браузере), и создавать новый элемент истории для такого взаимодействия было бы неуместно.

Однако в приложениях SPA также происходят многочисленные взаимодействия, управляемые AJAX, для которых уместно создать элемент истории. Существует специальный JavaScript API для работы с историей браузера, но этот API неудобен и с ним трудно работать, поэтому разработчики JavaScript часто его игнорируют.

Если вам доводилось случайно щелкнуть на кнопке возврата в приложении SPA, из-за чего все состояние приложения терялось и вам приходилось начинать работу заново, значит, вы уже сталкивались с этой проблемой.

В `htmx`, как и во фреймворках SPA, часто приходится явно работать с API истории просмотра. К счастью, так как `htmx` придерживается «родной» модели веб-среды, а также благодаря декларативной природе `htmx` в приложениях на ее основе работать с историей просмотра обычно намного проще.

Возьмем знакомую нам кнопку для загрузки контактов.

Листинг 43. Кнопка, проверенная временем

```
<button hx-get="/contacts" hx-target="#main">
 Get Contacts
</button>
```

Если щелкнуть на этой кнопке в текущей версии, кнопка прочитает контент из `/contacts` и загрузит его в элемент с идентификатором `main`, но не создаст новую запись в истории просмотра.

Если вам требуется создать запись в истории просмотра при выдаче запроса, добавьте к кнопке новый атрибут `hx-push-url`.

Листинг 44. Кнопка, проверенная временем, – теперь с историей!

```
<button hx-get="/contacts" hx-target="#main" hx-push-url="true"> ❶
 Get Contacts
</button>
```

❶ hx-push-url создает запись в истории просмотра по щелчку на кнопку.

Теперь по щелчку на кнопке путь `/contacts` помещается в адресную строку браузера, и для него создается запись в истории просмотра. Более того, если пользователь щелкнет на кнопке возврата, будет восстановлен исходный контент страницы вместе с исходным URL.

Имя атрибута `hx-push-url` выглядит немного странно, но оно происходит от JavaScript API `history.pushState()`. Концепция «проталкивания» (pushing) основана на том факте, что записи истории просмотра хранятся в стеке, так что, по сути, новые записи «проталкиваются» на вершину стека истории.

Благодаря этому относительно простому декларативному механизму `htmx` позволяет интегрироваться с кнопкой возврата способом, похожим на «нормальное» поведение HTML.

Но чтобы история работала действительно правильно, необходимо внести еще одно изменение: мы успешно поместили путь `/contacts` в адресную строку браузера, и кнопка возврата работает. Но что, если кто-то обновит браузер во время нахождения на странице `/contacts`?

В таком случае необходимо обработать «частичный» ответ `htmx` как «полностраничный» ответ, не использующий `htmx`. Для этого можно воспользоваться заголовками HTTP — эта тема будет более подробно рассмотрена позже.

Заключение

Наше быстрое знакомство с `htmx` подходит к концу. Мы рассмотрели лишь около десятка атрибутов из библиотеки, но уже получили некоторое представление о возможностях этих атрибутов. `Htmx` позволяет написать намного более сложное веб-приложение, чем с помощью базового HTML, и при этом дополнительная концептуальная нагрузка останется минимальной по сравнению с большинством решений на основе JavaScript.

`Htmx` призвана постепенно улучшать HTML как гипермедиа, причем делать это способом, соотносимым с базовым языком разметки. Как и у любого технического средства, у нее есть не только плюсы, но и минусы: оставаясь близкой к HTML, `htmx` не предоставляет разработчику той инфраструктуры, которая, по мнению многих, должна присутствовать «по умолчанию».

Оставаясь ближе к исходной модели веб-среды, `htmx` стремится выдержать баланс между простотой и функциональностью, доверяя другим библиотекам построение нетривиальных интерфейсных расширений на основе существующей веб-платформы. К счастью, `htmx` хорошо сочетается с другими библиотеками, и при необходимости подключить их достаточно просто.

Заметки об HTML: планирование бюджета для HTML

Тесная связь между контентом и разметкой означает, что создание качественной разметки HTML — весьма трудозатратное дело. При работе со многими сайтами дизайнеры, редко знакомые с HTML, отделены от разработчиков, которым нужно создать обобщенную систему, способную справиться с любым передаваемым контентом. Обычно такое разделение воплощается в системе управления контентом, или *CMS* (content management system). В результате разметка, адаптированная к контенту (что часто необходимо для продвинутого HTML), редко оказывается подходящей.

Более того, для интернационализированных сайтов внедрение контента на разных языках в одни и те же элементы может снизить качество разметки, так как в разных языках используются разные соглашения о стилях. Лишь немногие организации могут позволить себе такие затраты.

Поэтому мы не ожидаем, что каждый сайт будет содержать идеально оранизованный HTML. Важнее всего избежать *неверного* HTML — иногда лучше воспользоваться универсальным элементом, чем выбрать конкретный, но ошибочный вариант.

Но если вы располагаете соответствующими ресурсами и можете уделить больше внимания HTML, сайт получится более проработанным.

ПАТТЕРНЫ HTML

Итак, вы увидели, каким образом `htmx` расширяет HTML как гипермедиа. Пора применить эту библиотеку в деле. Используя `htmx`, мы используем гипермедиа: выдаем запросы HTTP и получаем ответы HTML. Но благодаря дополнительной функциональности, предоставляемой `htmx`, в нашем распоряжении появляется *более мощная платформа гипермедиа*, позволяющая создавать намного более сложные интерфейсы.

Решить многие проблемы взаимодействия с пользователем, например долгие циклы обратной связи или обновления страниц, можно без необходимости писать много кода JavaScript и создавать JSON API. Решение можно реализовать средствами гипермедиа с использованием основных гипермедиа-концепций ранней веб-среды.

Установка htmx

Прежде всего необходимо установить `htmx` в веб-приложение. Для этого следует загрузить исходный код и сохранить его локально в приложении, чтобы не зависеть от внешних систем. Чтобы загрузить последнюю версию `htmx`, введите в браузере адрес https://unpkg.com/htmx.org, с которого вы будете перенаправлены к исходному коду библиотеки.

Загруженный контент можно сохранить в файле `static/js/htmx.js` в нашем проекте.

Конечно, для установки `htmx` можно воспользоваться более функциональным менеджером пакетов JavaScript, таким как NPM (Node Package Manager) или `yarn`. Для этого следует сослаться на имя пакета `htmx.org` способом, подходящим для инструмента, который вы используете. Однако код `htmx` очень мал по размеру (примерно 12 Кбайт после сжатия и архивации) и свободен от зависимостей, так что его использование не требует сложных механизмов или систем сборки.

После того как библиотека `htmx` будет помещена в локальный каталог `/static/js` приложения, ее можно загрузить в приложение. Для этого следует включить следующий тег `script` в тег `head` в файле `layout.html`, в результате чего библиотека `htmx` станет доступной и активной для каждой страницы приложения.

Листинг 45. Установка htmx

```
<head>
 <script src="/js/htmx.js"></script>
 ...
</head>
```

Напомним, что файл `layout.html` представляет собой файл *макета*, который включается в большинство шаблонов. Он упаковывает содержимое таких шаблонов в стандартный HTML, включая элемент `head`, который используется здесь для установки `htmx`.

И это все! Простой тег `script` делает функциональность `htmx` доступной во всем приложении.

«AJAX-ификация» приложения

Приступим к решению практических задач с `htmx`, начав с использования механизма, называемого «усилением» (boosting). Это маленькая «волшебная» функция, для реализации которой вам почти не придется ничего делать — нужно лишь включить всего один атрибут `hx-boost`.

Когда вы добавляете в элемент атрибут `hx-boost` со значением `true`, тем самым вы «усиливаете» все якорные теги и элементы форм внутри этого элемента. В данном случае это означает, что `htmx` преобразует все якоря и формы из «обычных» гипермедиа-элементов управления в гипермедиа-элементы на основе AJAX. Вместо «обычных» запросов HTTP, заменяющих всю страницу, ссылки и формы теперь выдают запросы AJAX. Затем в ответ на эти запросы `htmx` заменяет внутренний контент тега `<body>` в теге `<body>` существующих страниц.

Это заметно ускоряет навигацию, потому что браузеру не приходится заново интерпретировать большинство тегов в ответе `<head>` и т. д.

Усиленные ссылки

Рассмотрим пример усиленной ссылки. Ниже приведена ссылка на гипотетическую страницу настроек веб-приложения. Так как в ней установлен атрибут `hx-boost="true"`, `htmx` приостанавливает обычное поведение ссылки с выдачей запроса к пути `/settings` и заменой всей страницы ответом. Вместо него `htmx` выдает запрос AJAX к `/settings`, получает результат и заменяет элемент `body` новым контентом.

Листинг 46. Усиленная ссылка

```
<a href="/settings" hx-boost="true">Settings</a> ❶
```

❶ Атрибут hx-boost делает ссылку поддерживаемой AJAX.

И чем это лучше, спросите вы? Мы выдаем запрос AJAX и просто заменяем все тело документа.

Разве это чем-то принципиально отличается от выдачи запроса по обычной ссылке? Да, на самом деле отличается: с усиленной ссылкой браузеру не нужно выполнять обработку, связанную с тегом `head`. Тег `head` часто содержит многочисленные скрипты и ссылки на файлы CSS. При использовании усиления заново обрабатывать эти ресурсы не нужно: скрипты и стили уже обработаны и будут применяться к новому контенту. Часто этот прием позволяет легко ускорить работу приложения гипермедиа.

Второй резонный вопрос: должен ли ответ быть закодирован в специальном формате для `hx-boost`? В конце концов, страница настроек обычно рендерит тег `html` с тегом `head` и т. д. Должны ли «усиленные» запросы обрабатываться каким-то особым образом?

Ответ — нет. Библиотека `htmx` достаточно «умна», чтобы извлечь только контент тега `body` для подстановки в новую страницу. Тег `head` в основном игнорируется: только тег `title`, если он присутствует, будет обработан. Это означает, что вам не нужно ничего специально делать на стороне сервера, чтобы рендерить шаблоны, которые могут обрабатываться `hx-boost`: просто верните стандартную разметку HTML для своей страницы, и все будет работать нормально.

Обратите внимание: усиленные ссылки (и формы) так же продолжают обновлять адресную строку и историю просмотра, как и обычные ссылки, так что пользователь сможет нажимать на кнопку возврата, копировать и вставлять URL (или «глубинные ссылки») и т. д. Ссылки ведут себя практически «как обычно», только работают быстрее.

Усиленные формы

Усиленные теги форм работают подобно усиленным якорным тегам: усиленная форма выдает запрос AJAX вместо обычного запроса браузера и заменяет все тело ответом.

Ниже приведен пример формы, которая отправляет сообщения конечной точке `/messages` с использованием запроса HTTP `POST`. При добавлении в нее атрибута `hx-boost` вместо стандартных запросов браузера будут выдаваться запросы AJAX.

Листинг 47. Усиленная форма

```
<form action="/messages" method="post" hx-boost="true"> ❶
 <input type="text" name="message" placeholder="Enter A Message...">
 <button>Post Your Message</button>
</form>
```

❶ Как и в случае со ссылкой, с атрибутом hx-boost эта форма будет работать с запросами AJAX.

Большое преимущество запросов на основе AJAX, используемых `hx-boost` (кроме отсутствия избыточной обработки `head`), заключается в том, что они предотвращают так называемое мерцание неоформленного контента , или FOUC (Flash Of Unstyled Content).

Мерцание неоформленного контента

> Ситуация, в которой браузер рендерит веб-страницу до получения всей информации о ее стилях. FOUC — раздражающее быстрое «мерцание» контента, получающего информацию о стилях и меняющего оформление. Данное явление нередко встречается при серфинге в интернете: текст, графика и другой контент только что загруженной страницы «прыгают» по странице при применении к ним стилей.

С `hx-boost` стилевое оформление сайта уже загружено перед получением нового контента, что исключает мерцание. В результате «усиленное» приложение кажется более плавным и в целом лучше смотрится.

Наследование атрибутов

Дополним предыдущий пример с усиленной ссылкой и добавим в него еще несколько усиленных ссылок. Ссылки будут вести на страницу `/contacts`, страницу `/settings` и страницу `/help`. Все эти ссылки усиливаются и ведут себя, как описано выше.

Выглядит излишне, верно? Нам приходится помечать три ссылки, расположенные рядом друг с другом, атрибутом `hx-boost="true"`, а это довольно странно.

Листинг 48. Группа усиленных ссылок

```
<a href="/contacts" hx-boost="true">Contacts</a>
<a href="/settings" hx-boost="true">Settings</a>
<a href="/help" hx-boost="true">Help</a>
```

`Htmx` поддерживает механизм наследования атрибутов, который позволяет устранить эту избыточность. В большинстве случаев в `htmx` атрибут, размещенный в родительском элементе, можно применить и к дочерним элементам. По этому принципу работает CSS, и эта идея вдохновила реализацию похожей функциональности «каскадных атрибутов» в `htmx`.

Чтобы избежать избыточности в нашем примере, добавим элемент `div`, включающий все ссылки, а затем «поднимем» атрибут `hx-boost` в родительский элемент `div`. Тогда избыточные атрибуты `hx-boost` можно будет убрать, при этом все ссылки останутся усиленными, так как они наследуют эту функциональность от родительского `div`.

Можно использовать любой допустимый элемент HTML, мы выбрали `div` просто по привычке.

Листинг 49. Усиление ссылок через родительский элемент

```
<div hx-boost="true"> ❶
  <a href="/contacts">Contacts</a>
  <a href="/settings">Settings</a>
  <a href="/help">Help</a>
</div>
```

❶ Атрибут hx-boost перемещен в родительский элемент div.

Теперь не нужно добавлять атрибут `hx-boost="true"` в каждую ссылку. Более того, можно добавить новые ссылки к уже существующим, и они тоже будут усилены автоматически.

Все это замечательно, но что, если внутри элемента с установленным атрибутом `hx-boost="true"` появится ссылка, которую *не нужно* усиливать? Хорошим примером может быть ссылка на загружаемый ресурс, например PDF-файл. Загрузку файлов не получится обработать запросами AJAX, так что, скорее всего, вы предпочтете, чтобы эта ссылка работала «как обычно», то есть выдавала для PDF-файла запрос полной страницы, чтобы браузер предложил сохранить файл в локальной системе пользователя.

В таких случаях достаточно переопределить родительское значение `hx-boost` значением `hx-boost="false"` в якорном теге, для которого требуется отключить усиление.

Листинг 50. Отключение усиления

```
<div hx-boost="true"> ❶
  <a href="/contacts">Contacts</a>
  <a href="/settings">Settings</a>
  <a href="/help">Help</a>
  <a href="/help/documentation.pdf" hx-boost="false">Download Docs</a> ❷
</div>
```

❶ Атрибут hx-boost остается в родительском div.
❷ Поведение усиления переопределяется для этой ссылки.

Здесь создается новая ссылка на PDF-файл документа, которая должна работать как обычная. Мы добавили в ссылку атрибут `hx-boost="false"`, и это объявление переопределяет `hx-boost="true"` в родительском `div`. В результате ссылка возвращается к обычному поведению загрузки файла.

Прогрессивное улучшение

У библиотеки `hx-boost` есть одна приятная особенность: она работает по принципу *прогрессивного улучшения* (progressive enchancement).

Прогрессивное улучшение

> Подход к разработке, цель которого — предоставить максимальный объем полезного контента и функциональности как можно большему количеству пользователей, обеспечивая при этом улучшенное взаимодействие для пользователей с последними версиями веб-браузеров.

Возьмем ссылки из приведенного выше примера. Что произойдет, если у пользователя будет отключен JavaScript?

Ничего страшного. Приложение продолжит работать, но будет выдавать обычные запросы HTTP вместо запросов HTTP на основе AJAX. Это означает, что веб-приложение будет работать у максимального количества пользователей; обладатели современных браузеров (или пользователи, не отключившие JavaScript) могут пользоваться преимуществами навигации в стиле AJAX, обеспечиваемыми `htmx`, но и у других приложение также будет работать.

Сравните поведение атрибута `hx-boost` из `htmx` с приложением SPA, активно использующим JavaScript: такое приложение часто *вообще* не будет работать с отключенным JavaScript. Использование фреймворков SPA часто сильно затрудняет применение подхода прогрессивного улучшения.

Это не значит, что все возможности `htmx` поддерживают прогрессивное улучшение. Конечно, можно реализовать функциональность, которая не предоставляет альтернативы «без JS» в `htmx`; более того, многое из того, что будет строиться далее в книге, относится именно к этой категории. Мы будем отмечать, совместима функциональность с прогрессивным улучшением или нет.

В конечном счете разработчик сам решает, компенсируются ли недостатки прогрессивного улучшения (более примитивный UX, ограниченный набор улучшений базового HTML) преимуществами для пользователей приложения.

Добавление hx-boost в Contact.app

В приложении для управления контактами это поведение «усиления» (boost) в `htmx` должно применяться... да везде.

Почему бы и нет?

И как этого добиться?

Очень просто (и такое решение часто встречается в веб-приложениях на основе `htmx`): можно добавить `hx-boost` в тег `body` шаблона `layout.html`, и больше ничего.

Листинг 51. Усиление во всем приложении contact.app

```
<html>
...
<body hx-boost="true"> ❶
...
</body>
</html>
```

❶ Теперь все ссылки и формы в приложении усилены!

Теперь все ссылки и формы в приложении будут использовать AJAX по умолчанию, из-за чего приложение кажется более динамичным. Рассмотрим ссылку для создания нового контакта, созданную на главной странице.

Листинг 52. Новая усиленная ссылка Add Contact

```
<a href="/contacts/new">Add Contact</a>
```

И хотя мы ничего не делали ни со ссылкой, ни с целевым URL на стороне сервера, теперь все ссылки «просто превращаются» в усиленные — с AJAX опыт взаимодействия становится динамичным, включая обновление истории просмотра, поддержку кнопки возврата и т. д. А если поддержка JavaScript отключена, приложение возвращается к обычному поведению ссылок.

И все это благодаря лишь одному атрибуту `htmx`.

Атрибут `hx-boost` удобен, но он отличается от других атрибутов своим «волшебным» свойством: одно незначительное изменение меняет поведение множества элементов страницы, превращая их в элементы на основе AJAX. Многие другие атрибуты `htmx` работают на более низком уровне и требуют более явных и точных инструкций. В целом это можно считать философией `htmx`: отдавать предпочтение явному перед скрытым и очевидному перед «волшебным».

Тем не менее атрибут `hx-boost` слишком хорош, чтобы жертвовать практичностью ради догмы, поэтому он включен в библиотеку.

Удаление контактов запросом HTTP DELETE

На следующем шаге работы с `htmx` напомним, что в Contact.app на странице редактирования контакта существует маленькая форма, используемая для удаления контакта.

Листинг 53. Простая форма HTML для удаления контакта

```
<form action="/contacts/{{ contact.id }}/delete" method="post">
  <button>Delete Contact</button>
</form>
```

Эта форма выдает запрос HTTP `POST`, например, к пути `/contacts/42/delete` для удаления контакта с идентификатором 42.

Мы уже отмечали, что у HTML есть одна неприятная особенность: запросы HTTP `DELETE` (а также `PUT` и `PATCH`) не могут выдаваться напрямую, хотя все они являются частью HTTP, а протокол HTTP *создавался непосредственно* для передачи HTML.

К счастью, `htmx` позволяет исправить этот недостаток.

«Правильное», ресурсно-ориентированное RESTful-решение заключается в том, чтобы вместо выдачи запроса HTTP `POST` к `/contacts/42/delete` выдать запрос HTTP `DELETE` к `/contacts/42`. Нам нужно удалить этот контакт. Контакт является ресурсом. URL этого ресурса имеет вид `/contacts/42`. Поэтому идеальное решение — запрос `DELETE` к `/contacts/42/`.

Чтобы реализовать эту возможность, добавим к кнопке Delete Contact («Удалить контакт») атрибут `hx-delete` библиотеки `htmx`.

Листинг 54. Кнопка на основе htmx для удаления контакта

```
<button hx-delete="/contacts/{{ contact.id }}">Delete Contact</button>
```

Теперь по щелчку на этой кнопке `htmx` выдает через AJAX запрос HTTP `DELETE` к заданному контакту.

Вот на что здесь стоит обратить внимание.

- Тег формы больше не нужен, поскольку сама кнопка заключает в себе действие гипермедиа, которое она выполняет над собой.
- Нет необходимости использовать громоздкий маршрут `"/contacts/{{ contact.id }}/delete"`. Вместо него можно взять простой маршрут `"/contacts/{{ contact.id }}`, так как мы выдаем запрос `DELETE`. `DELETE` помогает отличить запрос на обновление контакта от запроса на его удаление. При этом используются «родные» средства HTTP, предназначенные именно для этой цели.

Отметим, что здесь происходит нечто особенное: кнопка превращается в *элемент гипермедиа*. Для выдачи запроса HTTP уже не нужно включать кнопку в тег формы: это совершенно автономный, полнофункциональный элемент гипермедиа. В этом суть `htmx`: благодаря ему разработчик получает возможность превратить любой элемент управления в элемент гипермедиа — полноценную часть гипермедиа-управляемого приложения.

Также следует заметить, что, в отличие от приведенных выше примеров с `hx-boost`, это решение *не обеспечивает* изящной деградации (graceful degradation). Для этого следовало бы заключить кнопку в элемент `form`, а также обрабатывать

POST на стороне сервера. Чтобы не усложнять приложение, мы опустим это нетривиальное решение.

Обновление кода на стороне сервера

Мы обновили код на стороне клиента (если HTML можно считать кодом), так что теперь он выдает запрос DELETE к соответствующему URL, но работа еще не закончена. Так как мы обновили и маршрут, и используемый метод HTTP, придется обновить и реализацию на стороне сервера для обработки нового запроса HTTP.

Листинг 55. Исходная версия серверного кода для удаления контакта

```
@app.route("/contacts/<contact_id>/delete", methods=["POST"])
def contacts_delete(contact_id=0):
  contact = Contact.find(contact_id)
  contact.delete()
  flash("Deleted Contact!")
  return redirect("/contacts")
```

В обработчик необходимо внести два изменения: обновить маршрут и метод HTTP, используемый для удаления контактов.

Листинг 56. Обновленный обработчик с новым маршрутом и методом

```
@app.route("/contacts/<contact_id>", methods=["DELETE"]) ❶
def contacts_delete(contact_id=0):
  contact = Contact.find(contact_id)
  contact.delete()
  flash("Deleted Contact!")
  return redirect("/contacts")
```

❶ Обновленный путь и метод для обработчика.

Решение стало намного проще и намного чище.

Проблема в коде ответа

К сожалению, в нашем обновленном обработчике скрывается проблема: по умолчанию во Flask метод `redirect()` реагирует кодом ответа `302 Found`.

Согласно веб-документации MDN (Mozilla Developer Network) код ответа 302[1] означает, что *метод* HTTP запроса *останется неизменным* при выдаче перенаправленного запроса HTTP.

[1] https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/302

Сейчас из `htmx` выдается запрос `DELETE`, а затем Flask выполняет перенаправление к пути `/contacts`. В соответствии с этой логикой перенаправленный запрос HTTP все еще будет использовать метод `DELETE`. Следовательно, в текущем состоянии браузер выдаст к `/contacts` запрос `DELETE`.

Это определенно *не то*, что нам нужно: мы хотим, чтобы при перенаправлении HTTP выдавался запрос `GET`, то есть чтобы рассмотренный выше паттерн «`POST`/Redirect/`GET`» превратился в «`DELETE`/Redirect/`GET`».

К счастью, существует другой код ответа, `303 See Other`[1], который делает именно то, что требуется: при получении кода ответа перенаправления 303 браузер выдает запрос `GET` к новому местоположению. Следовательно, в контроллере должен использоваться код ответа 303.

Это несложно: у `redirect()` имеется второй параметр для передачи числового кода ответа, который нужно отправить.

Листинг 57. Обновленный обработчик с кодом ответа 303

```
@app.route("/contacts/<contact_id>", methods=["DELETE"])
def contacts_delete(contact_id=0):
  contact = Contact.find(contact_id)
  contact.delete()
  flash("Deleted Contact!")
  return redirect("/contacts", 303) ❶
```

❶ Используется код ответа 303.

В обновленной версии для удаления заданного контакта можно просто выдать запрос `DELETE` к тому же URL, который изначально использовался для обращения к контакту.

Так выглядит подход к удалению ресурсов, принятый для HTTP.

Выбор верного целевого элемента

Работа над обновлением кнопки удаления еще не закончена. Вспомните, что по умолчанию `htmx` выбирает «целевой» элемент, выдавший запрос, и помещает разметку HTML, возвращенную сервером, внутрь этого элемента. В текущей версии целевой является сама кнопка Delete Contact (Удалить контакт).

А это означает, что, поскольку перенаправление к URL `/contacts` приведет к повторному рендерингу всего списка контактов, этот список будет помещен *внутрь* кнопки Delete Contact.

[1] https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/303

Подобные случаи неверного выбора цели периодически встречаются при работе с `htmx` и могут приводить к довольно забавным ситуациям.

Проблема решается просто: укажите для кнопки явную цель — элемент `body`.

Листинг 58. Исправленная кнопка для удаления контакта на основе htmx

```
<button hx-delete="/contacts/{{ contact.id }}"
    hx-target="body"> ❶
 Delete Contact
</button>
```

❶ Цель для кнопки указана явно.

Кнопка работает, как предполагалось: по щелчку выдается запрос HTTP `DELETE` к URL текущего контакта на сервере, контакт удаляется, и происходит перенаправление к странице со списком контактов и флеш-сообщением.

Теперь-то все работает как надо?

Корректное обновление URL в адресной строке

...Почти.

Можно заметить, что по щелчку на кнопке, несмотря на перенаправление, в адресной строке выводится неверный URL. Он все еще указывает на `/contacts/{{ contact.id }}`. Дело в том, что мы не позаботились об обновлении URL: `htmx` просто выдает запрос `DELETE` и обновляет DOM ответом.

Как упоминалось выше, усиление с атрибутом `hx-boost` естественным образом обновляет адресную строку, моделируя поведение обычных якорных ссылок и форм, но в данном случае для выдачи `DELETE` создается гипермедиа-элемент управления — специальная кнопка. Требуется сообщить `htmx`, что URL этого запроса необходимо поместить в адресную строку.

Для этого следует добавить в кнопку атрибут `hx-push-url` со значением `true`.

Листинг 59. Удаление контакта с обновлением адресной строки

```
<button hx-delete="/contacts/{{ contact.id }}"
    hx-push-url="true" ❶
    hx-target="body">
 Delete Contact
</button>
```

❶ Дает команду htmx поместить URL перенаправления в адресную строку.

Вот *теперь* все работает как надо.

Мы создали кнопку, которая самостоятельно выдает надлежащим образом отформатированный запрос HTTP `DELETE` к верному URL с соответствующим обновлением пользовательского интерфейса и адресной строки. Для этого непосредственно в кнопку были добавлены три декларативных атрибута: `hx-delete`, `hx-target` и `hx-push-url`.

Нам пришлось сделать больше, чем при добавлении `hx-boost`, но явно прописанные подробности помогают увидеть, что делает кнопка как настраиваемый гипермедиа-элемент управления. Решение получается чистым; в нем используются встроенные возможности веб-среды как системы гипермедиа без нестандартных приемов работы с URL.

И еще кое-что...

Есть еще одна функция, которую можно добавить к кнопке Delete Contact: диалоговое окно подтверждения. Удаление контакта является деструктивной операцией, и в текущей реализации даже при случайном щелчке на кнопке Delete Contact приложение просто удалит контакт — возможно, к большому огорчению пользователя.

К счастью, в `htmx` существует простой механизм для включения подтверждений при деструктивных операциях: атрибут `hx-confirm`. Добавьте этот атрибут к элементу, укажите текст сообщения в качестве его значения, и перед выдачей запроса будет вызван метод JavaScript `confirm()`. Он открывает простое диалоговое окно, которое предлагает пользователю подтвердить выполняемое действие. Получается очень простой и удобный способ предотвращения случайностей.

В следующем примере реализовано подтверждение для операции удаления контактов.

Листинг 60. Подтверждение удаления

```
<button hx-delete="/contacts/{{ contact.id }}"
    hx-push-url="true"
    hx-confirm="Are you sure you want to delete this contact?" ❶
    hx-target="body">
  Delete Contact
</button>
```

❶ Пользователь получит сообщение, предлагающее подтвердить удаление.

Теперь, когда пользователь нажимает кнопку Delete Contact, он видит вопрос: Are you sure you want to delete this contact? (Вы уверены, что хотите удалить этот контакт?), и у него будет возможность отменить действие, если он нажал кнопку по ошибке. Очень хорошо.

Благодаря этому последнему изменению у нас появился довольно надежный механизм удаления контакта: в нем использованы правильные RESTful-

маршруты и методы HTTP, релизовано подтверждение удаления и убран лишний мусор, навязываемый обычно HTML, — и все это с использованием декларативных атрибутов в HTML и в рамках стандартной веб-модели гипермедиа.

Прогрессивное улучшение или нет?

Как уже говорилось, это решение не является прогрессивным улучшением веб-приложения. Если пользователь отключит JavaScript, кнопка Delete Contact перестанет работать. Чтобы старый механизм на основе форм работал в среде с отключенным JavaScript, придется потрудиться.

Прогрессивное улучшение — модная тема в веб-разработке, с разными субъективными мнениями и множеством точек зрения. Как и почти со всеми библиотеками JavaScript, с `htmx` можно создать приложение, не функционирующее при отсутствии JavaScript. Сохранение поддержки для клиентов с недоступным JavaScript увеличивает объем работы и повышает сложность приложения. Следует точно определить, насколько важна поддержка клиентов без JavaScript, еще до того, как вы начнете использовать `htmx` или любой другой фреймворк JavaScript для расширения возможностей веб-приложения.

Проверка адресов электронной почты

Перейдем к очередному улучшению нашего приложения. Большую часть работы любого приложения занимает проверка данных, отправляемых серверу: проверка формата и уникальности адресов электронной почты, допустимости числовых значений, дат и т. д.

Сейчас в нашем приложении реализована минимальная проверка данных, и выполняется она исключительно на стороне сервера, а в случае обнаружения ошибок выдается сообщение.

Мы не будем подробно рассказывать, как работает проверка данных в объектах моделей, но вспомните код обновления контакта из главы 3.

Листинг 61. Проверка данных на стороне сервера при обновлении контакта

```
def contacts_edit_post(contact_id=0):
  c = Contact.find(contact_id)
  c.update(request.form['first_name'], request.form['last_name'],
request.form['phone'], request.form['email'])
  if c.save(): ❶
    flash("Updated Contact!")
    return redirect("/contacts/" + str(contact_id))
  else:
    return render_template("edit.html", contact=c) ❷
```

❶ Пытается сохранить контакт.
❷ В случае неудачи форма выводится заново с сообщением об ошибках.

Итак, мы пытаемся сохранить контакт, и если метод `save()` возвращает `true`, происходит перенаправление на страницу с подробной информацией о контакте. Если метод `save()` не возвращает `true`, это означает, что при проверке данных произошла ошибка; вместо перенаправления заново рендерится HTML для редактирования контакта. Тем самым пользователю предоставляется возможность исправить ошибки, информация о которых отображается рядом с введенными данными. Посмотрим внимательнее на разметку HTML для поля ввода адреса электронной почты.

Листинг 62. Сообщения об ошибках проверки данных

```
<p>
  <label for="email">Email</label>
  <input name="email" id="email" type="text" placeholder="Email" value="{{
contact.email }}">
  <span class="error">{{ contact.errors['email'] }}</span> ❶
</p>
```

❶ Выводит любые ошибки, относящиеся к полю ввода адреса электронной почты.

Мы создаем текстовую метку, поле ввода с типом `text` и небольшой фрагмент HTML для вывода сообщений об ошибках, относящихся к полю ввода адреса электронной почты. Когда шаблон рендерится на сервере, обнаруженные ошибки будут выведены в элементе `span` красным цветом.

ЛОГИКА ПРОВЕРКИ НА СТОРОНЕ СЕРВЕРА

Сейчас в классе контакта присутствует логика, которая проверяет, существуют ли другие контакты с таким же адресом электронной почты, и при их обнаружении добавляет ошибку в модель контакта, так как в основе данных не должны присутствовать повторяющиеся адреса. Это очень распространенный пример проверки данных: адреса обычно уникальны, и добавление двух контактов с одним адресом почти всегда указывает на ошибку пользователя.

И снова мы не будем углубляться в принципы проверки данных в моделях, но почти все фреймворки на стороне сервера предоставляют средства для такой проверки и сбора ошибок, выводимых пользователю. Такая инфраструктура типична для фреймворков Web 1.0 на стороне сервера.

Сообщение об ошибке, выводимое при попытке сохранения контакта с повторяющимся адресом электронной почты.

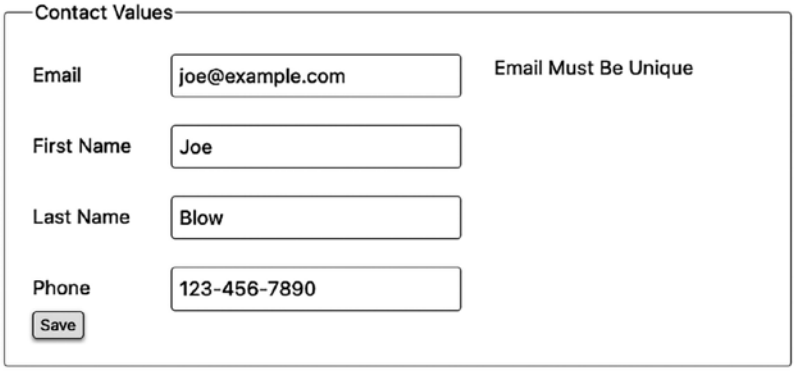


Рис. 5. Ошибка проверки адреса электронной почты

Это решение создается с использованием базового HTML и функциональности Web 1.0 и работает нормально.

Тем не менее в текущей версии приложения есть две проблемы.

- Во-первых, не проверяется формат адреса: вместо адреса электронной почты можно ввести любую последовательность символов, и при условии уникальности она будет принята системой.
- Во-вторых, уникальность адреса проверяется только при отправке данных: если пользователь ввел дублирующийся адрес, он не узнает об этом, пока не будут заполнены все поля. Это может быть весьма неприятно, если пользователь случайно вводит контакт повторно и узнает об этом только после ввода всей информации.

Обновление типа поля ввода

Для решения первой проблемы существует «чистый» механизм HTML: в HTML 5 поддерживаются поля ввода с типом `email`. Все, что для этого нужно, — сменить тип `text` на `email`, и браузер сам проверит, что введенное значение соответствует формату электронной почты.

Листинг 63. Переход на тип поля ввода email

```
<p>
  <label for="email">Email</label>
  <input name="email" id="email" type="email" placeholder="Email" value="{{
contact.email }}"> ❶
  <span class="error">{{ contact.errors['email'] }}</span>
</p>
```

❶ Изменение атрибута type на email гарантирует, что введенные значения будут соответствовать формату адресов электронной почты.

После внесения этого изменения будет происходить следующее: если пользователь вводит значение, не являющееся допустимым адресом электронной почты, браузер выдаст сообщение об ошибке и предложит ввести верный адрес электронной почты в этом поле.

Таким образом, простое изменение одного атрибута, внесенное непосредственно в HTML, повышает эффективность проверки и решает первую проблему.

ПРОВЕРКИ НА СТОРОНЕ СЕРВЕРА И НА СТОРОНЕ КЛИЕНТА

У веб-разработчиков со стажем приведенный выше код вызывает скрежет зубовный: проверка выполняется *на стороне клиента*. Другими словами, мы полагаемся на то, что браузер обнаружит неверно оформленный адрес и исправит пользователя. К сожалению, проверка на стороне клиента ненадежна: в браузере может присутствовать ошибка, которая позволит пользователю обойти код проверки. Или, что еще хуже, пользователь может быть злоумышленником, который постарается полностью обойти проверку: например, воспользуется консолью разработчика для редактирования HTML.

Это извечный риск веб-разработки: всем проверкам, выполняемым на стороне клиента, нельзя доверять полностью, и действительно важные проверки должны *выполняться повторно* на стороне сервера. В приложениях гипермедиа эта проблема не настолько критична, как в одностраничных приложениях, поскольку приложения гипермедиа серверно-ориентированы, и все же при создании приложений о ней следует помнить.

Встроенная проверка данных

Хотя нам удалось немного повысить эффективность проверки, пользователь все равно должен отправить данные формы, чтобы получить обратную связь о повторяющихся адресах. Затем можно воспользоваться `htmx` для улучшения качества взаимодействия с пользователем.

Лучше, если пользователь увидит сообщение о повторяющихся адресах сразу же после заполнения поля. Оказывается, поля ввода выдают событие `change`; кроме того, событие `change` *по умолчанию инициирует* выдачу запроса для полей ввода в `htmx`. Воспользовавшись этим, можно реализовать следующее поведение: когда пользователь вводит адрес, немедленно направить запрос к серверу, проверить введенный адрес и при необходимости выдать сообщение об ошибке.

Вспомните текущую разметку HTML для поля ввода адреса электронной почты.

Листинг 64. Исходная конфигурация поля ввода адреса электронной почты

```
<p>
  <label for="email">Email</label>
  <input name="email" id="email" type="email" placeholder="Email" value="{{
contact.email }}"> ❶
  <span class="error">{{ contact.errors['email'] }}</span> ❷
</p>
```

❶ Поле ввода, которое должно выдавать запрос HTTP для проверки адреса.
❷ Тег span, в который помещается сообщение об ошибке (если оно есть).

Итак, нам нужно добавить атрибут `hx-get` к этому полю ввода, чтобы оно выдавало запрос HTTP `GET` к заданному URL для проверки адреса электронной почты. Целевым для сообщений об ошибках, возвращаемых от сервера, должен быть выбран тег `span`, следующий за полем ввода.

Для этого внесем в HTML следующее изменение.

Листинг 65. Обновленная разметка HTML

```
<p>
  <label for="email">Email</label>
  <input name="email" id="email" type="email"
     hx-get="/contacts/{{ contact.id }}/email" ❶
     hx-target="next .error" ❷
     placeholder="Email" value="{{ contact.email }}"> ❶
  <span class="error">{{ contact.errors['email'] }}</span>
</p>
```

❶ Выдает запрос HTTP GET к конечной точке email для контакта.
❷ Выбирает целевым следующий элемент с классом error.

Обратите внимание: в атрибуте `hx-target` используется *относительный позиционный* селектор `next`. Это особенность `htmx` и расширение для обычного CSS. `Htmx` поддерживает префиксы для поиска целевых элементов *относительно* текущего элемента.

Относительные позиционные выражения в HTMX

- `next` — найти следующий целевой элемент прямым сканированием DOM (например, `next .error`).
- `previous` — найти предыдущий целевой элемент обратным сканированием DOM (например, `previous .alert`).
- `closest` — найти целевой элемент сканированием родительских элементов (например, `closest table`).

- `find` — найти целевой элемент сканированием дочерних элементов (например, `find span`).
- `this` — целевым элементом является текущий элемент (используется по умолчанию).

Использование относительных позиционных выражений позволяет избежать явного включения идентификаторов элементов и пользоваться локальной структурой HTML.

Таким образом, в нашем примере с добавлением атрибутов `hx-get` и `hx-target` каждый раз, когда пользователь изменяет значение поля ввода (напомним, что `change` *по умолчанию* инициирует запросы для полей ввода в `htmx`), выдается запрос HTTP `GET` к заданному URL. Обнаруженные ошибки (при наличии) будут загружены в `span`.

Проверка адресов электронной почты на стороне сервера

На следующем шаге рассматривается реализация на стороне сервера. Мы добавим еще одну конечную точку, которая чем-то напоминает конечную точку `edit`: она будет искать контакт по его ID, закодированному в URL. Однако на этот раз обновляться будет только адрес электронной почты контакта, и, очевидно, сохранять его не нужно! Вместо этого для него будет вызываться метод `validate()`.

Метод будет проверять, что адреса электронной почты уникальны, и т. д. В этой точке можно напрямую вернуть любые ошибки, связанные с адресом электронной почты, или пустую строку при отсутствии ошибок.

Листинг 66. Код конечной точки проверки адресов электронной почты

```
@app.route("/contacts/<contact_id>/email", methods=["GET"])
def contacts_email_get(contact_id=0):
  c = Contact.find(contact_id) ❶
  c.email = request.args.get('email') ❷
  c.validate() ❸
  return c.errors.get('email') or "" ❹
```

❶ Ищет контакт по его ID.

❷ Обновляет его адрес электронной почты (так как выполняется запрос GET, мы используем свойство args вместо свойства form).

❸ Проверяет контакт.

❹ Возвращает строку с ошибками, относящимися к полю электронной почты, или пустую строку при отсутствии ошибок.

В этом небольшом фрагменте кода на стороне сервера реализуется следующая функциональность: когда пользователь вводит адрес электронной почты и переходит к следующему полю ввода, он сразу получает оповещение, если введенный адрес уже используется.

Заметим, что проверка адреса электронной почты также выполняется при отправке обновляемого контакта, что исключает риск случайного появления дублей адресов: при помощи `htmx` мы просто сделали так, чтобы дубли обнаруживались раньше.

Также стоит заметить, что эта конкретная проверка *обязательно должна* выполняться на стороне сервера: невозможно проверить, что адрес уникален среди всех контактов, если у вас нет доступа к хранилищу данных записей. Это еще одна особенность приложений гипермедиа, упрощающая работу с ними: так как проверка выполняется на стороне сервера, вам доступны все данные, которые могут понадобиться для любых проверок.

Здесь мы снова хотим подчеркнуть, что описанное выше взаимодействие осуществляется полностью в рамках модели гипермедиа: мы используем декларативные атрибуты и обмениваемся данными гипермедиа с сервером, причем механизм обмена очень похож на механизм работы ссылок или форм. При этом нам удается радикально повысить качество взаимодействия с пользователем.

Как еще улучшить взаимодействие с пользователем

Хотя мы и не добавляли много кода, нам удалось создать довольно сложный пользовательский интерфейс — по крайней мере по сравнению с приложениями на основе простого HTML. Но если вы работали в более продвинутых приложениях SPA, то, вероятно, видели, что в них поле для адреса электронной почты (или похожей информации) проверяется прямо в процессе ввода.

И такая интерактивность возможна только при использовании сложных комплексных фреймворков JavaScript, не так ли?

Вообще-то нет.

Оказывается, эту функциональность можно реализовать в `htmx` при помощи одних лишь атрибутов HTML.

Собственно, все, что для этого нужно, — изменить триггер инициирования запроса. В настоящее время для полей ввода используется условие по умолчанию, то есть событие `change`. Чтобы данные проверялись прямо во время ввода, также необходимо обрабатывать событие `keyup`.

Листинг 67. Выдача запроса по событию keyup

```
<p>
  <label for="email">Email</label>
  <input name="email" id="email" type="email"
     hx-get="/contacts/{{ contact.id }}/email"
     hx-target="next .error"
     hx-trigger="change, keyup" ❶
     placeholder="Email" value="{{ contact.email }}">
  <span class="error">{{ contact.errors['email'] }}</span>
</p>
```

❶ К событию change явно добавляется событие keyup.

Всего одно крошечное изменение, но теперь каждый раз, когда пользователь вводит символ, приложение выдает запрос и проверяет адрес электронной почты. Все просто.

Устранение дребезга запросов

Просто, да. Но, наверное, не то, что вам нужно: выдача запроса при каждом событии `keyup` приведет к большим тратам ресурсов и теоретически может создать лишнюю нагрузку на сервер. Запрос должен выдаваться только тогда, когда пользователь делает недолгую паузу при вводе. Это называется «устранение дребезга» (debouncing) при вводе: запросы выдаются не сразу, а с небольшой задержкой.

`Htmx` поддерживает модификатор `delay` для триггеров запросов, которые позволяют устранять дребезг запроса за счет добавления задержки перед отправкой запроса. Если в пределах этой задержки происходит другое событие того же типа, `htmx` не выдает запрос и таймер обнуляется.

Это именно то, что нужно для поля ввода адреса электронной почты: если пользователь вводит адрес, мы не прерываем его, но как только он делает паузу или покидает поле, выдается запрос.

Добавим задержку в 200 миллисекунд для события `keyup` — этого достаточно, чтобы обнаружить, что пользователь перестал вводить адрес.

Листинг 68. Устранение дребезга для события keyup

```
<p>
  <label for="email">Email</label>
  <input name="email" id="email" type="email"
     hx-get="/contacts/{{ contact.id }}/email"
     hx-target="next .error"
     hx-trigger="change, keyup delay:200ms" ❶
```

```
    placeholder="Email" value="{{ contact.email }}">
  <span class="error">{{ contact.errors['email'] }}</span>
</p>
```

❶ Чтобы устранить дребезг для keyup, добавляем модификатор delay.

После этого приложение перестает выдавать поток запросов на проверку данных при вводе. Вместо этого оно ждет, пока пользователь сделает небольшую паузу, а затем выдает запрос. Нагрузка на сервер снижается, а качество взаимодействия сохраняется.

Игнорирование клавиш, не влияющих на ввод

Осталось решить последнюю проблему с событием `keyup`: в текущей версии запрос выдается независимо от того, *какие* клавиши были нажаты, даже если эти клавиши никак не влияют на вводимое значение (например, клавиши управления курсором). Намного лучше выдавать запрос только при изменении содержимого поля ввода.

Оказывается, `htmx` поддерживает этот конкретный паттерн — в нем к событиям применяется модификатор `changed`. (Не путайте с событием `change`, которое инициируется DOM для элементов ввода.)

При добавлении модификатора `changed` к событию `keyup` поле ввода не выдает запросы на проверку данных, если только событие `keyup` не обновляет значения поля ввода.

Листинг 69. Запросы отправляются только при изменении значения поля ввода

```
<p>
  <label for="email">Email</label>
  <input name="email" id="email" type="email"
     hx-get="/contacts/{{ contact.id }}/email"
     hx-target="next .error"
     hx-trigger="change, keyup delay:200ms changed" ❶
     placeholder="Email" value="{{ contact.email }}">
  <span class="error">{{ contact.errors['email'] }}</span>
</p>
```

❶ Чтобы предотвратить лишние запросы, выдавать их только при фактическом изменении значения поля ввода.

Получается привлекательная и эффективная разметка HTML, которая позволяет делать то, для чего, по мнению многих разработчиков, потребуется сложное решение на стороне клиента.

Благодаря всего трем атрибутам и простой новой конечной точке на стороне сервера мы улучшили пользовательский опыт в нашем веб-приложении. А что еще лучше, любые правила проверки адреса электронной почты, добавленные на стороне сервера, с этой моделью вообще работают *автоматически*: так как гипермедиа используется в качестве механизма коммуникации, нет необходимости обеспечивать синхронизацию моделей на стороне клиента и на стороне сервера.

Прекрасный пример возможностей архитектуры гипермедиа!

Еще одно улучшение: разбивка на страницы

Ненадолго оставим страницу редактирования контактов и улучшим корневую страницу приложения. Эта страница, доступная по пути `/contacts`, рендерит шаблон `index.html`.

В текущей версии Contact.app не поддерживает разбивку на страницы: если база данных содержит 10 000 контактов, то все 10 000 контактов будут выведены на корневой странице. Вывод такого объема данных может замедлить работу браузера (и сервера), поэтому во многих веб-приложениях для работы с наборами данных такого объема используется концепция «страничного вывода»: отображается только одна «страница» с небольшим количеством записей и средствами для перемещения между страницами набора данных.

Исправим наше приложение так, чтобы в нем выводились только 10 контактов, а также ссылки для перехода к следующей и предыдущей странице, если в базе данных содержится более 10 контактов.

Первое, что мы для этого сделаем, — добавим в шаблон `index.html` простой виджет страничного вывода.

В шаблон добавляются две ссылки с условиями:

- если текущая страница не является первой, то добавляется ссылка на предыдущую страницу;
- если текущий набор содержит 10 контактов, то добавляется ссылка на следующую страницу.

Конечно, такой виджет далеко не идеален: нормальная реализация должна выводить общее количество страниц и поддерживать более четкие условия перехода по страницам. Кроме того, существует вероятность, что следующая страница содержит 0 результатов, так как мы не проверяем их общее количество, но для нашего простого приложения хватит и этого.

Ниже приведен соответствующий фрагмент шаблона Jinja из `index.html`.

Листинг 70. Добавление ссылок для страничного вывода списка контактов

```
<div>
  <span style="float: right"> ❶
    {% if page > 1 %}
     <a href="/contacts?page={{ page - 1 }}">Previous</a> ❷
    {% endif %}
    {% if contacts|length == 10 %}
     <a href="/contacts?page={{ page + 1 }}">Next</a> ❸
    {% endif %}
  </span>
</div>
```

❶ Включает под таблицей новый элемент div для навигационных ссылок.

❷ Если текущая страница не первая, включает якорный тег с номером страницы, уменьшенным на 1.

❸ Если текущая страница содержит 10 контактов, включает якорный тег со ссылкой на следующую страницу, номер которой увеличивается на 1.

Обратите внимание на использование специального синтаксиса фильтров Jinja `contacts|length` для вычисления длины списка контактов. Подробное описание синтаксиса фильтров выходит за рамки книги, но если кратко, то выражение обращается к свойству `contacts.length` и сравнивает его с 10.

Ссылки готовы, теперь можно заняться реализацией разбивки на страницы на стороне сервера.

Параметр запроса `page` используется для кодирования состояния страничного вывода в пользовательском интерфейсе. Таким образом, в обработчике нужно проверить параметр `page` и передать его модели в виде целого числа, чтобы модель знала, какую страницу контактов нужно вернуть.

Листинг 71. Включение страничного вывода в обработчик запросов

```
@app.route("/contacts")
def contacts():
  search = request.args.get("q")
  page = int(request.args.get("page", 1)) ❶
  if search is not None:
    contacts_set = Contact.search(search)
  else:
    contacts_set = Contact.all(page) ❷
  return render_template("index.html", contacts=contacts_set, page=page)
```

❶ Получает значение параметра page; если page не передается, по умолчанию используется значение 1.

❷ Передает page модели при загрузке всех контактов, чтобы модель знала, какую страницу из 10 контактов необходимо вернуть.

Решение довольно прямолинейное: нужно лишь получить еще один параметр (по аналогии с параметром `q`, который передавался при поиске контактов ранее), преобразовать его в целое число, а затем передать модели `Contact`, чтобы она знала, какую страницу нужно вернуть. И это маленькое изменение — все, что нам нужно; теперь в нашем веб-приложении появился простейший механизм страничного вывода. Удивительно, но он уже использует AJAX благодаря добавлению `hx-boost` в приложение. Да, вот так просто!

Загрузка по щелчку

Механизм страничного вывода хорошо подходит для базовых веб-приложений и часто применяется в интернете. Однако у него есть недостатки: каждый раз, когда вы щелкаете на кнопке Next (Далее) или Previous (Назад), вы получаете новую страницу контактов и полностью теряете весь контекст предыдущей страницы.

Иногда предпочтительнее более совершенный UI-паттерн страничного вывода. Например, вместо загрузки новой страницы элементов и замены текущих элементов лучше *встроить* следующую страницу элементов внутрь текущей страницы, после текущей группы элементов.

Это распространенный UX-паттерн «загрузки по щелчку», встречающийся в более современных веб-приложениях.

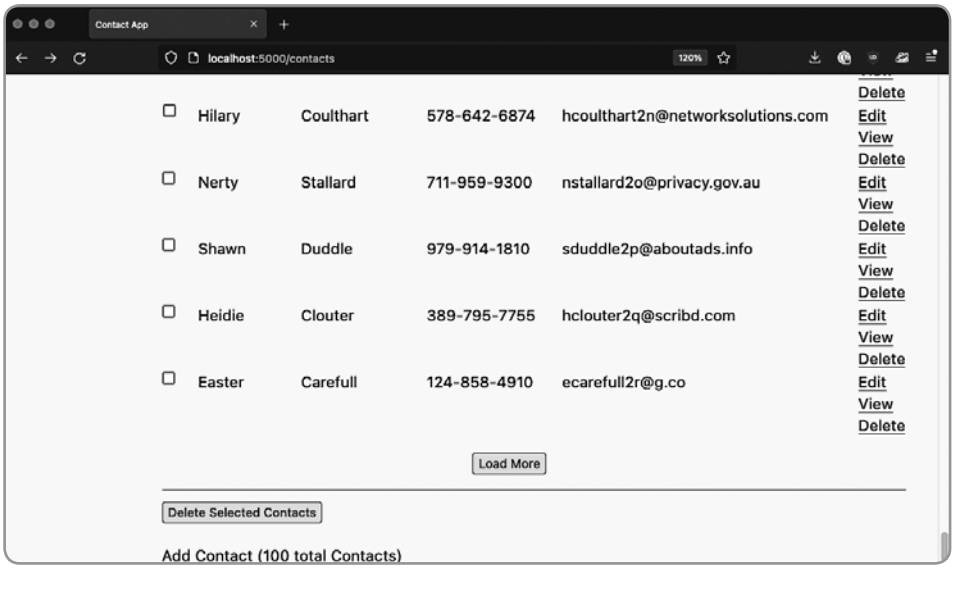


Рис. 6. Пользовательский интерфейс загрузки по щелчку

Кнопка загружает следующую группу контактов прямо на текущей странице, а не открывает новую страницу. Это позволяет открывать контакты в контексте текущей страницы, но при этом работать с ними как в стандартном пользовательском интерфейсе со страничным выводом.

Посмотрим, как реализовать этот паттерн UX в `htmx`. Все на удивление просто: нужно взять существующую ссылку Next (Далее) и немного переработать ее. И для этого не понадобится ничего, кроме нескольких атрибутов `htmx`! Фактически нужно создать кнопку, которая по щелчку присоединяет строки со следующей страницы контактов к текущей таблице (вместо того, чтобы рендерить всю таблицу заново). Для этого можно добавить в таблицу новую строку с такой кнопкой.

Листинг 72. Переход на «загрузку по щелчку»

```
    <tbody>
    {% for contact in contacts %}
      <tr>
        <td>{{ contact.first }}</td>
        <td>{{ contact.last }}</td>
        <td>{{ contact.phone }}</td>
        <td>{{ contact.email }}</td>
        <td><a href="/contacts/{{ contact.id }}/edit">Edit</a> <a
href="/contacts/{{ contact.id }}">View</a></td>
      </tr>
    {% endfor %}
    {% if contacts|length == 10 %} ❶
      <tr>
        <td colspan="5" style="text-align: center">
          <button hx-target="closest tr" ❷
              hx-swap="outerHTML"  ❸
              hx-select="tbody > tr" ❹
              hx-get="/contacts?page={{ page + 1 }}">
           Load More
          </button>
        </td>
      </tr>
   {% endif %}
   </tbody>
```

❶ Кнопка Load More (Загрузить еще) отображается, только если текущая страница содержит 10 контактов.

❷ Целевой назначается ближайшая внешняя строка.

❸ Вся строка заменяется ответом от сервера.

❹ Выбираются строки таблицы из ответа.

Разберем каждый атрибут более подробно.

Сначала атрибут `hx-target` используется для выбора «ближайшего» элемента `tr`, то есть ближайшей строки *родительской* таблицы.

Затем *вся* строка заменяется контентом, полученным от сервера.

После этого из ответа извлекаются только элементы `tr`. Текущий элемент `tr` заменяется новым набором элементов `tr`, содержащих дополнительные контакты, а также в случае необходимости — новую кнопку Load More (Загрузить еще), которая ссылается на *следующую* страницу. Для этого используется селектор CSS `tbody > tr`, который гарантирует, что извлекаться будут только строки из тела таблицы. Тем самым предотвращается, например, включение строк из заголовка таблицы.

В конце выдается запрос HTTP `GET` к URL, который выводит следующую страницу контактов; он такой же, как для ссылки Next, которую мы рассматривали выше.

На удивление, новая функциональность не требует никаких изменений на стороне сервера. Это объясняется гибкостью, которую предоставляет `htmx` в отношении обработки ответов от сервера.

Итак, всего четыре атрибута — и мы реализовали современный пользовательский интерфейс «загрузки по щелчку» на основе `htmx`.

Бесконечная прокрутка

Еще один распространенный паттерн при работе с большими наборами данных —«Бесконечная прокрутка» (Infinite Scroll). В этом паттерне при появлении последнего элемента таблицы или списка элементов в результате прокрутки дополнительные элементы загружаются и присоединяются к списку или таблице.

Вообще говоря, такое поведение больше подходит в ситуациях, когда пользователь просматривает посты в социальной сети, чем в приложении для управления контактами. Тем не менее для полноты картины и просто для демонстрации возможностей `htmx` мы реализуем и этот паттерн.

Оказывается, код «загрузки по щелчку» достаточно легко переработать для этой цели: если задуматься, бесконечная прокрутка использует логику «загрузки по щелчку», только вместо загрузки по возникновению события `click` загрузка должна осуществляться при «появлении» элемента в области просмотра браузера.

И снова нам повезло: `htmx` предоставляет синтетическое (нестандартное) событие DOM `revealed`, которое может использоваться в сочетании с атрибутом `hx-trigger` для выдачи запроса, когда элемент становится видимым.

Преобразуем кнопку в `span` и воспользуемся возможностями, которые предоставляет это событие.

Листинг 73. Переход на «бесконечную прокрутку»

```
{% if contacts|length == 10 %} ❶
  <tr>
    <td colspan="5" style="text-align: center">
      <span hx-target="closest tr" ❶
         hx-trigger="revealed" ❷
         hx-swap="outerHTML"
         hx-select="tbody > tr"
         hx-get="/contacts?page={{ page + 1 }}">Loading More...
</span>
    </td>
  </tr>
{% endif %}
```

❶ Элемент button преобразован в span, так как пользователь не будет щелкать на нем.

❷ Запрос инициируется, когда элемент становится видимым, то есть появляется в области просмотра.

Все, что необходимо для преобразования «запроса по щелчку» в «бесконечную прокрутку», — изменить тип элемента на `span`, а затем добавить триггер для события `revealed`.

Простота переключения на бесконечную прокрутку показывает, как `htmx` добавляет HTML универсальности: всего несколько атрибутов позволили кардинально улучшить результат, получаемый с применением гипермедиа. И снова при этом использовались преимущества RESTful-модели веб-среды. Несмотря на новое поведение, мы по-прежнему обмениваемся с сервером данными гипермедиа, и ответы JSON API в этом обмене не участвуют.

Как, собственно, и проектировалась веб-среда.

Заметки об HTML: будьте осторожны с модальными окнами и display: none

Добавляя модальные окна, подумайте как следует. Этот элемент стал очень популярным, почти стандартным во многих современных веб-приложениях.

К сожалению, модальные окна не лучшим образом сочетаются с инфраструктурой веб-среды и вводят состояние на стороне клиента, которое может быть трудно (а то и невозможно) интегрировать с гипермедиа.

Модальные окна могут безопасно использоваться для представлений, которые не являются ресурсами или не соответствуют сущностям предметной области:

- оповещений;
- диалоговых окон подтверждения;
- форм для создания/обновления сущностей.

В остальных случаях лучше рассмотреть такие альтернативы, как встроенное редактирование или отдельная страница вместо модальной.

Будьте внимательны при использовании `display: none;`. Дело в том, что эта конструкция не является чисто косметической — она также удаляет элементы из дерева доступности и фокуса клавиатуры. Иногда это делается для того, чтобы создать для контента как визуальное, так звуковое представление. Если вы хотите скрыть элемент из визуального представления, не скрывая его от вспомогательных технологий (например, если элемент содержит информацию, которая передается на уровне стиля), можно воспользоваться вспомогательным классом.

```
.vh {
      clip: rect(0 0 0 0);
      clip-path: inset(50%);
      block-size: 1px;
      inline-size: 1px;
      overflow: hidden;
      white-space: nowrap;
}
```

`vh` — сокращение от visually hidden, то есть «визуально скрытое». Этот класс использует различные методы и обходные решения, чтобы гарантировать, что никакой браузер не удалит функцию элемента.

ДРУГИЕ ПАТТЕРНЫ HTMX

Активный поиск

Пока работа над Contact.app идет довольно успешно: мы создали маленькое веб-приложение, но значительно улучшенное по сравнению с простым приложением на основе HTML. Мы добавили корректно работающую кнопку удаления контакта, организовали динамическую проверку ввода и рассмотрели разные способы добавления страничного вывода в приложение. Как мы уже говорили, многие веб-разработчики считают, что реализовать эти возможности можно только с помощью скриптов JavaScript, но мы все сделали на относительно чистом HTML, используя только атрибуты `htmx`.

Со временем мы *добавим* в приложение скриптовый код на стороне клиента: при всей своей эффективности гипермедиа нельзя назвать *всемогущей*, и иногда скрипты оказываются лучшим (или единственным) способом достичь заданной цели. А пока посмотрим, чего можно добиться с гипермедиа.

Первая нетривиальная возможность, которую мы реализуем на основе `htmx`, — паттерн «Активный поиск». Он означает, что когда пользователь вводит текст в поле поиска, результаты поиска динамически изменяются. Этот паттерн приобрел популярность, когда его внедрила компания Google, и теперь реализуется во многих приложениях.

Чтобы реализовать паттерн «Активный поиск», воспользуемся средствами, тесно связанными с механикой проверки адресов электронной почты из предыдущей главы. Если подумать, у этих двух возможностей много общего: в обоих случаях при вводе текста в поле должен выдаваться запрос, а при получении ответа должен обновляться другой элемент. Конечно, реализации на стороне сервера будут сильно различаться, но код фронтенда будет одинаковым благодаря единому подходу «выдать запрос по событию и заменить элемент на экране», принятому в `htmx`.

Текущий пользовательский интерфейс поиска

Напомним, как сейчас выглядит поле поиска нашего приложения.

Листинг 74. Форма поиска

```
<form action="/contacts" method="get" class="tool-bar">
  <label for="search">Search Term</label>
  <input id="search" type="search" name="q" value="{{ request.args.get('q')
or '' }}"/> ❶
  <input type="submit" value="Search"/>
</form>
```

❶ Параметр, который будет использоваться для поиска кодом на стороне клиента.

Напомним, что код на стороне сервера ищет параметр `q`, и если он присутствует, ищет этот текст в контактах.

В этой версии пользователь должен нажать Ввод, когда фокус установлен на поле ввода поискового запроса, или кнопку Search (Поиск). Оба способа инициируют событие `submit` для формы, указывая ей выдать запрос HTTP `GET` и заново отрендерить всю страницу.

В текущей версии благодаря `hx-boost` форма использует запрос AJAX для этого `GET`, но мы еще не реализовали нужное поведение поиска при вводе.

Добавление активного поиска

Чтобы добавить поведение активного поиска, присоединим к полю ввода поискового запроса несколько атрибутов `htmx`. Пока текущая форма останется без изменений, с `action` и `method`, чтобы нормальное поведение поиска работало даже в том случае, если у пользователя не включен JavaScript. Тем самым реализация активного поиска представляет собой хороший пример «прогресивного улучшения».

Итак, в дополнение к обычному поведению форм, запрос HTTP `GET` *также* должен выдаваться по событию `keyup`. Этот запрос должен выдаваться к тому же URL, что и обычная отправка данных формы. Наконец, это должно происходить только при возникновении небольшой паузы при вводе.

Как уже было сказано, эта функциональность очень похожа на механизм проверки адреса электронной почты. Собственно, атрибут `hx-trigger` с маленькой 200-миллисекундной задержкой можно скопировать прямо из примера проверки адреса, чтобы запрос выдавался только при наступлении паузы при вводе.

Это еще один пример того, как при использовании `htmx` снова и снова встречаются уже знакомые паттерны.

Листинг 75. Добавление поведения активного поиска

```
<form action="/contacts" method="get" class="tool-bar">
  <label for="search">Search Term</label>
  <input id="search" type="search" name="q" value="{{ request.args.get('q')
or '' }}" ❶
     hx-get="/contacts" ❷
     hx-trigger="search, keyup delay:200ms changed"/> ❸
  <input type="submit" value="Search"/>
</form>
```

❶ Оставляем исходные атрибуты, чтобы поиск работал, даже если JavaScript недоступен.

❷ Выдает запрос GET к тому же URL, что и форма.

❸ Атрибут hx-trigger почти полностью совпадает с примером проверки адреса электронной почты.

В атрибут `hx-trigger` было внесено небольшое изменение: событие `change` заменено событием `search`. Событие `search` выдается при очистке поиска или при нажатии клавиши Enter. Это нестандартное событие, но от его включения вреда не будет. Основная функциональность обеспечивается вторым событием-триггером, `keyup`. Как и в примере с электронной почтой, этот триггер срабатывает с задержкой: модификатор `200ms` устраняет дребезг запросов от поля ввода и предотвращает отправку запросов на сервер при каждом событии `keyup`.

Выбор правильного целевого элемента

Результат близок к желаемому, но еще нужно задать правильный целевой элемент. Вспомните, что целью по умолчанию для элемента является он сам. В текущей версии запрос HTTP `GET` будет выдаваться к пути `/contacts`; он (опять-таки в текущей версии) вернет полный документ HTML с результатами поиска, после чего весь документ будет вставлен *внутрь* HTML поля поиска.

И это полная бессмыслица: элементы `input` не могут содержать в себе HTML. Браузер в этом случае просто проигнорирует запрос `htmx` на включение HTML ответа в `input`. Таким образом, в текущей версии при отправке ввода пользователем будет выдан запрос (его можно увидеть в браузерной консоли разработчика), но, к сожалению, для пользователя ничего не изменится.

Что же выбрать целью для обновления, чтобы решить эту проблему? В идеале это должны быть фактические результаты; нет никаких причин обновлять заголовок или поле ввода поиска, к тому же это вызовет раздражающее мерцание при смене фокуса.

Атрибут `hx-target` позволяет сделать именно это. Воспользуемся им, чтобы выбрать целью тело результатов — элемент `tbody` в таблице контактов.

Листинг 76. Добавление поведения активного поиска

```
<form action="/contacts" method="get" class="tool-bar">
  <label for="search">Search Term</label>
  <input id="search" type="search" name="q" value="{{ request.args.get('q')
or '' }}"
     hx-get="/contacts"
     hx-trigger="search, keyup delay:200ms changed"
     hx-target="tbody"/> ❶
  <input type="submit" value="Search"/>
</form>
<table>
  ...
  <tbody>
   ...
  </tbody>
</table>
```

❶ Выбирает целевым тег tbody на странице.

Так как на странице присутствует всего один элемент `tbody`, мы можем воспользоваться общим селектором CSS `tbody`, и `htmx` выберет целью таблицу на странице.

Если теперь ввести что-нибудь в поле поиска, мы увидим результаты: выдается запрос, и результаты вставляются в документ в элементе `tbody`. К сожалению, возвращаемый контент все еще представляет собой полный документ HTML.

Мы приходим к ситуации «двойного рендера», когда весь документ вставлен *внутрь* другого элемента, так что все средства навигации, шапки, подвалы и пр. будут заново отрендерены внутри этого элемента. Это один из примеров проблемы неверного выбора цели, о котором мы говорили.

К счастью, проблема решается относительно легко.

Выбор части контента

Можно воспользоваться тем же инструментом, который пригодился при реализации функциональности «загрузки по щелчку» и «бесконечной прокрутки»: атрибутом `hx-select`. Напомним, что атрибут `hx-select` позволяет выбрать части ответа, которые вас интересуют, при помощи селектора CSS.

Следовательно, к полю ввода можно добавить следующее определение.

Листинг 77. Использование атрибута hx-select для реализации активного поиска

```
<input id="search" type="search" name="q" value="{{ request.args.get('q') or
'' }}"
   hx-get="/contacts"
   hx-trigger="change, keyup delay:200ms changed"
```

```
   hx-target="tbody"
   hx-select="tbody tr"/> ❶
```

❶ Добавление атрибута hx-select для выбора строк таблицы в теге tbody ответа.

Впрочем, это не единственное решение проблемы, и в данном случае оно и не самое эффективное. Реализуем другой вариант: изменим код *на стороне сервера* нашего гипермедиа-управляемого приложения, чтобы он возвращал *только нужный контент HTML*.

Заголовки запросов HTTP в htmx

В этом разделе рассматривается другое, более эффективное решение для ситуации, в которой нам нужна только *часть* HTML вместо полного документа. В текущей версии сервер создает полный документ HTML в качестве ответа, а затем на стороне клиента HTML фильтруется до нужных частей. Делать это несложно и вообще может быть необходимо, если вы не контролируете сторону сервера или не можете легко изменить ответ.

Но так как в нашем приложении используется «полностековая» разработка (то есть мы контролируем как фронтенд, так и бэкенд и можем легко изменять эти части), появляется другой вариант: можно изменить ответы сервера, чтобы он возвращал только нужный контент, и избавиться от необходимости выполнять фильтрацию на стороне клиента.

Такое решение оказывается более эффективным, так как мы не возвращаем контент, сопутствующий интересующей нас части; при этом снижается загрузка канала связи, а также экономятся ресурсы процессора и памяти на стороне сервера. Рассмотрим, как возвращать разный контент HTML в зависимости от контекстной информации, отправляемой `htmx` вместе с запросами HTTP.

Еще раз приведем текущий код логики поиска на стороне сервера.

Листинг 78. Поиск на стороне сервера

```
@app.route("/contacts")
def contacts():
  search = request.args.get("q")
  if search is not None:
    contacts_set = Contact.search(search) ❶
  else:
    contacts_set = Contact.all()
  return render_template("index.html", contacts=contacts_set) ❷
```

❶ Здесь выполняется логика поиска.

❷ Шаблон index.html рендерится заново во всех сценариях.

Как изменить эту логику? Она должна рендерить два разных фрагмента разметки HTML *в зависимости от условий*:

- если это «стандартный» запрос всей страницы, шаблон index.html рендерится как обычно. Собственно, для «стандартных» запросов ничего меняться не должно;
- однако для запросов активного поиска должен рендериться только контент, находящийся внутри `tbody`, то есть строки таблицы.

Итак, нужно точно определить, к какому из двух типов относится запрос к URL `/contact`, чтобы знать, какой контент следует рендерить.

Оказывается, `htmx` поможет различить эти два случая; нужно только добавить соответствующие *заголовки запросов* HTTP при выдаче этих запросов. Заголовки запросов относятся к функциональности HTTP; они позволяют клиентам (например, веб-браузерам) связывать с запросами пары метаданных «имя/значение», которые помогут серверу понять, что же именно запрашивает клиент.

Ниже приведен пример (некоторых) заголовков, выдаваемых браузером FireFox при запросе по адресу https://hypermedia.systems.

Листинг 79. Заголовки HTTP

```
GET / HTTP/2
Host: hypermedia.systems
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:103.0)
Gecko/20100101 Firefox/103.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*
/*;q=0.8
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.5
Cache-Control: no-cache
Connection: keep-alive
DNT: 1
Pragma: no-cache
```

`Htmx` пользуется этой возможностью HTTP и добавляет дополнительные заголовки и, следовательно, дополнительный *контекст* к выдаваемым запросам HTTP. Это позволяет проанализировать заголовки и решить, какая логика должна выполняться на сервере и какой ответ HTML должен быть отправлен клиенту.

Ниже перечислены заголовки HTTP, которые `htmx` включает в запросы HTTP:

- `HX-Boosted` — строка true, если запрос выдается через элемент, использующий `hx-boost`;
- `HX-Current-URL` — текущий URL браузера;

- `HX-History-Restore-Request` — строка true, если запрос предназначен для восстановления истории после промаха в локальном кэше истории;
- `HX-Prompt` — содержит ответ пользователя на `hx-prompt`;
- `HX-Request` — всегда содержит true для запросов на основе `htmx`;
- `HX-Target` — идентификатор целевого элемента (если существует);
- `HX-Trigger-Name` — имя элемента, инициировавшего запрос (если существует);
- `HX-Trigger` — идентификатор элемента, инициировавшего запрос (если существует).

В этом списке заголовков выделяется последний: идентификатор, то есть `search` для поля ввода поиска. Следовательно, при поступлении запроса от поля ввода поиска (с идентификатором `search`) заголовок `HX-Trigger` будет содержать значение `search`.

Добавим в контроллер условную логику, которая ищет этот заголовок и, если он содержит значение `search`, рендерит только строки вместо всего шаблона index.html.

Листинг 80. Обновление поиска на стороне сервера

```
@app.route("/contacts")
def contacts():
  search = request.args.get("q")
  if search is not None:
    contacts_set = Contact.search(search)
    if request.headers.get('HX-Trigger') == 'search': ❶
     # TODO: здесь отрендерить только строки таблицы ❷
  else:
    contacts_set = Contact.all()
  return render_template("index.html", contacts=contacts_set) ❷
```

❶ Если заголовок запроса HX-Trigger содержит «search», необходимо выполнить иное действие.

❷ Нужно понять, как отрендерить только строки таблицы.

Итак, как же отрендерить только строки таблицы для результата?

Факторизация шаблонов

Мы пришли к одному из распространенных паттернов `htmx`: *факторизации* (разложению на составляющие) шаблонов на стороне сервера. Это означает, что шаблоны необходимо разделить, чтобы они могли вызываться из разных контекстов. В данном случае строки таблицы требуется выделить в отдельный шаблон, который мы назовем `rows.html`. Он будет включен в исходный шаблон

index.html, а также будет рендериться в контроллере сам по себе, когда в ответ должны быть включены только строки таблицы для запросов активного поиска.

Сейчас таблица в файле index.html выглядит так:

Листинг 81. Таблица с информацией о контактах

```
  <table>
    <thead>
    <tr>
      <th>First</th> <th>Last</th> <th>Phone</th> <th>Email</th> <th>
</th>
    </tr>
    </thead>
    <tbody>
    {% for contact in contacts %}
      <tr>
        <td>{{ contact.first }}</td>
        <td>{{ contact.last }}</td>
        <td>{{ contact.phone }}</td>
        <td>{{ contact.email }}</td>
        <td><a href="/contacts/{{ contact.id }}/edit">Edit</a>
          <a href="/contacts/{{ contact.id }}">View</a></td>
      </tr>
    {% endfor %}
    </tbody>
  </table>
```

Цикл for в этом шаблоне генерирует все строки в итоговом контенте, выпускаемом шаблоном index.html. Нужно вынести цикл for (а следовательно, и создаваемые им строки) в *отдельный файл шаблона*, чтобы небольшая часть HTML могла рендериться независимо от index.html.

Новому шаблону присваивается имя rows.html.

Листинг 82. Новый файл rows.html

```
{% for contact in contacts %} ❷
  <tr>
    <td>{{ contact.first }}</td>
    <td>{{ contact.last }}</td>
    <td>{{ contact.phone }}</td>
    <td>{{ contact.email }}</td>
    <td><a href="/contacts/{{ contact.id }}/edit">Edit</a>
      <a href="/contacts/{{ contact.id }}">View</a></td>
  </tr>
{% endfor %}
```

При помощи этого шаблона можно рендерить только элементы tr для заданного набора контактов.

Конечно, этот контент все еще должен включаться в шаблон `index.html`: иногда будет рендериться целая страница, а иногда только строки таблицы. Чтобы шаблон `index.html` рендерился нормально, добавим шаблон `rows.html` директивой Jinja `include` в расположение, куда должен вставляться контент из `rows.html`.

Листинг 83. Добавление нового файла

```
  <table>
    <thead>
    <tr>
      <th>First</th>
      <th>Last</th>
      <th>Phone</th>
      <th>Email</th>
      <th></th>
    </tr>
    </thead>
    <tbody>
    {% include 'rows.html' %} ❶
    </tbody>
  </table>
```

❶ Эта директива «добавляет» файл rows.html, подставляя его содержимое в текущий шаблон.

Пока все неплохо: страница `/contacts` рендерится нормально, как и до выделения строк из шаблона `index.html`.

Использование нового шаблона

Последним шагом факторизации шаблонов станет изменение контроллера, чтобы при ответе на запрос активного поиска в нем использовался новый файл шаблона `rows.html`.

Так как `rows.html` — всего лишь еще один шаблон, принципиально не отличающийся от `index.html`, все, что нужно сделать, — вызвать функцию `render_template` для `rows.html` вместо `index.html`. Функция рендерит *только* контент строк вместо всей страницы.

Листинг 84. Обновление поиска на стороне сервера

```
@app.route("/contacts")
def contacts():
  search = request.args.get("q")
  if search is not None:
    contacts_set = Contact.search(search)
    if request.headers.get('HX-Trigger') == 'search':
```

```
    return render_template("rows.html", contacts=contacts_set) ❶
  else:
    contacts_set = Contact.all()
  return render_template("index.html", contacts=contacts_set)
```

❶ Рендерит новый шаблон в случае активного поиска.

Теперь при выдаче запроса активного поиска вместо всего документа HTML мы получаем только часть HTML — строки таблицы для контактов, соответствующих поиску. Затем эти строки вставляются в тег `tbody` страницы `index` без необходимости использовать `hx-select` или иную обработку на стороне клиента.

При этом поиск на основе форм продолжает работать. Строки условно рендерятся только в том случае, если поле ввода `search` выдает запрос HTTP через `htmx`. Такой подход также служит примером прогрессивного улучшения приложения.

ЗАГОЛОВКИ HTTP И КЭШИРОВАНИЕ

У представленного решения, использующего заголовки для определения возвращаемого контента, есть одна неочевидная особенность. Речь идет о функциональности кэширования, встроенной в HTTP. В своем обработчике запроса мы возвращаем разный контент в зависимости от значения заголовка HX-Trigger. Если вы собираетесь пользоваться кэшированием HTTP, может возникнуть ситуация, в которой кто-то выдает *обычный* запрос (например, на обновление страницы), а из кэша будет возвращен контент `htmx`, что приведет к получению пользователем части контента.

Проблема решается использованием заголовка HTTP Response Vary и перечислением заголовков `htmx`, используемых для определения возвращаемого контента. Полное объяснение кэширования HTTP выходит за рамки книги, но в MDN есть хорошая статья на эту тему[1]; кроме того, этот вопрос подробно рассмотрен в документации `htmx`[2].

Обновление адресной строки при помощи hx-push-url

У текущей реализации активного поиска имеется один недостаток по сравнению с обычной отправкой данных формы: версия с отправкой данных обновляет адресную строку браузера, включая в нее условие поиска. Таким образом, например, если ввести в поле поиска строку «joe», URL в адресной строке браузера будет выглядеть примерно так:

[1] https://developer.mozilla.org/en-US/docs/Web/HTTP/Caching

[2] https://htmx.org/docs/#caching

Листинг 85. Обновленный адрес после отправки формы

```
https://example.com/contacts?q=joe
```

Это удобно: можно создать закладку для поиска или скопировать URL и отправить его кому-то другому. Получателю остается щелкнуть на ссылке, чтобы выполнить точно такой же поиск. Кроме того, эта функциональность связана с концепцией истории просмотра в браузере: щелчок на кнопке возврата вернет вас к URL, с которого вы пришли. Если вы выполнили поиск по двум условиям и хотите возвратиться к первому, просто нажмите кнопку возврата, и браузер «вернется» к предыдущему поиску.

В текущей версии во время активного поиска адресная строка браузера не обновляется. Таким образом, пользователи не получают ссылки, которые можно копировать и вставлять, а вы не получаете записи в истории просмотра, что означает отсутствие поддержки кнопки возврата. К счастью, вы уже знаете, как решить эту проблему: при помощи атрибута `hx-push-url`.

Атрибут `hx-push-url` позволяет указать `htmx`: «Протолкни URL этого запроса в адресную строку браузера». Возможно, термин «протолкнуть» (push) выглядит немного странно, но он используется в API истории просмотра; дело в том, что в API история браузера моделируется в форме стека: при переходе к новой странице адрес «проталкивается» в стек элементов истории, а при щелчке кнопки возврата адрес «выталкивается» из стека истории.

Итак, чтобы обеспечить поддержку истории для активного поиска, потребуется совсем немного: присвоить атрибуту `hx-push-url` значение `true`.

Листинг 86. Обновление URL во время активного поиска

```
<input id="search" type="search" name="q" value="{{ request.args.get('q') or
'' }}"
   hx-get="/contacts"
   hx-trigger="change, keyup delay:200ms changed"
   hx-target="tbody"
   hx-push-url="true"/> ❶
```

❶ С добавлением атрибута hx-push-url со значением true htmx будет обновлять URL при выдаче запроса.

Теперь при отправке запросов активного поиска обновляется URL в адресной строке браузера: в него включается запрос, как и при отправке формы.

Возможно, такое поведение покажется вам *нежелательным*. Например, вы считаете, что обновление адресной строки и создание элементов истории просмотра для каждого активного поиска только сбивает пользователя с толку. И это нормально: просто уберите атрибут `hx-push-url`, и все вернется к желаемому по-

ведению. Библиотека htmx достаточно гибкая, чтобы обеспечить нужный опыт взаимодействия, оставаясь в границах декларативной модели HTML.

Добавление индикатора запроса

К реализации паттерна «Активный поиск» остается добавить последний штрих — индикатор запроса, чтобы пользователь понимал, что ведется поиск. В текущей версии ничто не указывает на то, что функциональность активного поиска обрабатывает запрос. Если поиск займет какое-то время, пользователь может подумать, что активный поиск не работает. Добавляя индикатор запроса, мы сообщаем пользователю, что гипермедиа-приложение занято и ему придется подождать (хочется надеяться, не слишком долго!) завершения запроса.

Для поддержки индикаторов запросов htmx использует атрибут hx-indicator. Атрибут получает — да, вы правильно догадались — селектор CSS, который указывает на индикатор заданного элемента. Индикатором может быть что угодно, но обычно используется графика с анимацией (например, файл gif или svg), например объект, который вращается или как-то иначе показывает, что в приложении «что-то происходит».

Добавим спиннер для активного поиска.

Листинг 87. Добавление индикатора запроса при поиске

```
<input id="search" type="search" name="q" value="{{ request.args.get('q') or
'' }}"
   hx-get="/contacts"
   hx-trigger="change, keyup delay:200ms changed"
   hx-target="tbody"
   hx-push-url="true"
   hx-indicator="#spinner"/> ❶
<img id="spinner" class="htmx-indicator" src="/static/img/spinning-
circles.svg" alt="Request In Flight..."/> ❷
```

❶ Атрибут hx-indicator указывает на изображение индикатора после ввода.

❷ Индикатор реализуется в виде svg-файла с вращающимся кругом, которому назначен класс htmx-indicator.

Индикатор добавляется непосредственно за полем ввода. Тем самым он визуально связывается с элементом, выдающим запрос, и пользователю становится проще понять, что здесь происходит что-то заслуживающее внимания.

Приложение работает, но как htmx заставляет индикатор появляться и исчезать? Обратите внимание: в теге img индикатора указан класс htmx-indicator — класс CSS, который автоматически внедряется в страницу библиотекой htmx. Класс по умолчанию назначает уровень непрозрачности 0, так что элемент скрывается и не влияет на макет страницы.

Когда инициируется запрос `htmx`, указывающий на этот индикатор, к индикатору добавляется другой класс, `htmx-request`, который переводит уровень непрозрачности в 1. Таким образом, в качестве индикатора можно использовать практически все что угодно; по умолчанию этот объект будет скрыт. Затем, при выдаче запроса, он появится на экране. Все это делается при помощи стандартных классов CSS, позволяющих управлять переходами и даже механизмом отображения индикатора (например, можно использовать `display` вместо `opacity`).

ИСПОЛЬЗУЙТЕ ИНДИКАТОРЫ ЗАПРОСОВ!

Индикаторы запросов – важный UX-элемент любого распределенного приложения. Печально, что со временем браузеры стали пренебрегать встроенными индикаторами запросов, и еще печальнее, что индикаторы запросов не являются частью JavaScript AJAX API.

Не пренебрегайте этим важным элементом приложения. Когда вы работаете с приложением локально, может показаться, что запросы выполняются мгновенно, но в реальных условиях они могут занимать некоторое время из-за сетевой задержки. Часто бывает полезно воспользоваться браузерными средствами разработчика, которые позволяют регулировать время отклика локального браузера. Это поможет лучше представить, что увидят реальные пользователи, и покажет, с какими индикаторами пользователю будет проще понять, что происходит.

После добавления индикатора запросов у нас получился достаточно продвинутый интерфейс, по сравнению с простым HTML, и при этом построенный полностью в виде гипермедиа-управляемой функциональности. В этом интерфейсе не приходится работать ни с JSON, ни с JavaScript. А еще наша реализация обладает таким достоинством, как прогрессивное улучшение; приложение будет работать и у клиентов, у которых поддержка JavaScript отключена.

Отложенная загрузка

Разобравшись с активным поиском, перейдем к другому улучшению: *отложенной*, или *ленивой*, *загрузке* (lazy loading). Этот термин означает, что загрузка некоторого контента откладывается до того момента, когда в нем возникнет необходимость. Обычно отложенная загрузка используется для повышения производительности: она позволяет избежать затрат ресурсов, необходимых для получения данных, пока в этих данных не возникнет прямой необходимости.

Добавим в Contact.app счетчик общего количества контактов, прямо под таблицей контактов. Таким образом, в приложении появляется потенциально затрат-

ная операция, на примере которой можно продемонстрировать, как добавить отложенную загрузку в `htmx`.

Сначала обновим код сервера в обработчике запроса `/contacts` для получения общего количества контактов. Счетчик будет передаваться шаблону для рендеринга новой разметки HTML.

Листинг 88. Добавление счетчика в UI

```
@app.route("/contacts")
def contacts():
  search = request.args.get("q")
  page = int(request.args.get("page", 1))
  count = Contact.count() ❶
  if search is not None:
    contacts_set = Contact.search(search)
    if request.headers.get('HX-Trigger') == 'search':
      return render_template("rows.html", contacts=contacts_set,
page=page, count=count)
  else:
    contacts_set = Contact.all(page)
  return render_template("index.html", contacts=contacts_set, page=page,
count=count) ❷
```

❶ Получает общее количество контактов из модели контактов.

❷ Счетчик передается шаблону index.html для использования при рендеринге.

Как и в остальном коде приложения, мы сосредоточимся на аспекте *гипермедиа* Contact.app и опустим подробности того, как работает метод `Contact.count()`. Достаточно знать следующее:

- метод возвращает общее количество контактов в основе данных;
- он может работать медленно (в контексте нашего примера).

Затем добавим в файл `index.html` разметку HTML, которая использует новые данные и выводит рядом со ссылкой Add Contact (Добавить контакт) сообщение с общим количеством пользователей. HTML будет выглядеть так:

Листинг 89. Добавление счетчика контактов в приложение

```
<p>
  <a href="/contacts/new">Add Contact</a> <span>({{ count }} total
Contacts)</span> ❶
</p>
```

❶ Простой элемент span с текстом, сообщающим общее количество контактов.

Очень просто, не так ли? Теперь рядом со ссылкой для добавления новых контактов пользователи будут видеть их общее количество; это даст им пред-

ставление о размере базы данных контактов. Подобные методы быстрой разработки — одно из преимуществ традиционной разработки веб-приложений.

Вот как выглядит новая функциональность в приложении:

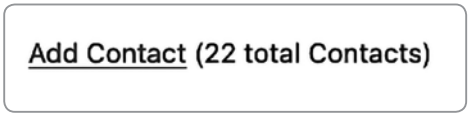


Рис. 7. Вывод общего количества контактов

Прекрасно.

Конечно, как вы наверняка уже догадались, все далеко не идеально. После внедрения новой функциональности вы начинаете получать жалобы от пользователей о том, что приложение «тормозит». Как и все хорошие разработчики, сталкивающиеся с проблемами производительности, вы не пытаетесь угадать причину, а проводите профилирование производительности.

Как ни странно, выясняется, что проблема возникает из-за безобидного вызова `Contacts.count()`, который занимает около 1,5 секунды. К сожалению, по причинам, о которых слишком долго рассказывать, сократить время загрузки невозможно, как и кэшировать результат.

Остаются два варианта:

- удалить счетчик контактов;
- придумать другое решение проблемы с производительностью.

Допустим, счетчик для вас слишком важен и отказаться от него нельзя. Придется решать проблему с производительностью средствами `htmx`.

Извлечение затратного кода

Первым шагом в реализации паттерна «Отложенная загрузка» становится извлечение затратного кода, то есть вызова `Contacts.count()`, из обработчика запроса к конечной точке `/contacts`.

Поместим этот вызов функции в отдельный обработчик запроса HTTP в виде новой конечной точки HTTP, доступной по адресу `/contacts/count`. Новая конечная точка вообще не должна рендерить шаблон: ее единственная задача — отрендерить маленький фрагмент текста в теге `span` — 22 total Contacts (Всего контактов: 22).

Новый код приведен в следующем листинге.

Листинг 90. Извлечение затратного кода

```
@app.route("/contacts")
def contacts():
  search = request.args.get("q")
  page = int(request.args.get("page", 1)) ❶
  if search is not None:
    contacts_set = Contact.search(search)
    if request.headers.get('HX-Trigger') == 'search':
      return render_template("rows.html", contacts=contacts_set,
page=page)
  else:
    contacts_set = Contact.all(page)
  return render_template("index.html", contacts=contacts_set, page=page) ❷

@app.route("/contacts/count")
def contacts_count():
  count = Contact.count() ❸
  return "(" + str(count) + " total Contacts)" ❹
```

❶ В этом обработчике не вызывается Contacts.count().

❷ Счетчик больше не передается шаблону для рендеринга в обработчике /contacts.

❸ Мы создаем новый обработчик по пути /contacts/count, который выполняет затратное вычисление.

❹ Возвращает строку с общим количеством контактов.

Мы убрали причину проблемы с производительностью из кода обработчика `/contacts`, который рендерит главную таблицу контактов, и создали новую конечную точку HTTP, которая выполнит затратную операцию по созданию строки за нас.

Теперь необходимо перенести контент из нового обработчика в `span`. Как уже говорилось, по умолчанию `htmx` помещает весь контент, полученный для запроса, в `innerHTML` этого элемента. И это именно то, что нам нужно: мы хотим получить текст и поместить его в `span`. Таким образом, можно просто задать атрибут `hx-get` для `span`, указать ссылку на новый путь — и результат будет достигнут.

Однако напомним, что по умолчанию запрос для элемента `span` в `htmx` инициируется *событием* `click`. Но это совершенно не то, что нам нужно! Запрос должен инициироваться немедленно в момент загрузки страницы.

Можно добавить атрибут `hx-trigger`, чтобы сменить триггер запросов для элемента и указать событие `load`.

Событие `load` — специальное событие, которое инициируется `htmx` для всего контента, загружаемого в DOM. Присваивая `hx-trigger` значение `load`, мы указываем, что `htmx` должен выдать запрос `GET` при загрузке элемента `span` на странице.

Обновленный код шаблона выглядит так:

Листинг 91. Добавление элемента со счетчиком контактов в приложение

```
<p>
  <a href="/contacts/new">Add Contact</a> <span hx-get="/contacts/count"
hx-trigger="load"></span> ❶
</p>
```

❶ Выдает запрос GET к /contacts/count при возникновении события load.

Обратите внимание: в исходном состоянии элемент `span` пуст: мы удалили из него контент, предоставляя возможность заполнить его запросу к `/contacts/count`.

Посмотрите — страница `/contacts` снова работает быстро! Когда вы переходите к странице, кажется, что она загружается практически мгновенно, — и профилирование показывает, что загрузка действительно ускорилась. Почему? Потому что мы вынесли затратные вычисления во вторичный запрос, и исходный запрос быстрее завершает загрузку.

«Прекрасно, — скажете вы, — но счетчик появляется на странице только через секунду или две». Верно, но пользователя не всегда интересует общее количество контактов. Он может просто зайти на страницу, чтобы найти, отредактировать или добавить пользователя. В таких случаях общее количество контактов знать хорошо, но не обязательно. Откладывая вычисление счетчика подобным образом, мы даем пользователю возможность работать с приложением, пока выполняются затратные вычисления.

Да, общее время получения всей информации на экране осталось прежним — и даже чуть увеличилось, так как теперь для получения всей информации потребуются два запроса HTTP. Однако *субъективная производительность* с точки зрения конечного пользователя заметно повысилась: пользователь может приступить к работе практически сразу, даже если какая-то информация не будет доступна немедленно.

Отложенная загрузка — отличный инструмент, который очень полезно держать под рукой при оптимизации производительности веб-приложений.

Добавление индикатора

К недостаткам текущей реализации можно отнести полное отсутствие информации о том, что запрос счетчика обрабатывается. Данные просто появляются в какой-то момент после завершения запроса.

Такая ситуация оставляет желать лучшего. Нам нужен идентификатор наподобие того, который был добавлен в примере с активным поиском. Собствен-

но, можно использовать тот же спиннер, скопировав его в новую разметку HTML.

Однако на этот раз используется одноразовый запрос, и после его завершения индикатор не понадобится. А значит, повторять решение из примера с активным поиском неэффективно. Напомним, что в том случае мы поместили индикатор *после* `span` и воспользовались атрибутом `hx-indicator` для создания ссылки на него.

В новой ситуации индикатор используется только один раз, и его можно включить в контент `span`. Когда запрос завершается, контент из ответа будет помещен в `span`, а индикатор заменится вычисленным счетчиком. Оказывается, `htmx` позволяет размещать индикаторы с классом `htmx-indicator` внутри элементов, выдающих запросы на основе `htmx`. При отсутствии атрибута `hx-indicator` эти внутренние индикаторы будут отображаться на время обработки запроса.

Итак, добавим спиннер из примера с активным поиском как исходный контент элемента `span`.

Листинг 92. Добавление индикатора для отложенной загрузки контента

```
<span hx-get="/contacts/count" hx-trigger="load">
 <img id="spinner" class="htmx-indicator" src="/static/img/spinning-
circles.svg"/> ❶
</span>
```

❶ Вот и индикатор.

Теперь при загрузке страницы значение счетчика не появляется как по волшебству, а выводится спиннер, который сигнализирует, что происходит что-то важное. Так намного лучше!

По сути все, что для этого потребовалось сделать, — скопировать и вставить индикатор из примера с активным поиском в `span`. И снова мы видим, как `htmx` предоставляет гибкие, легко комбинируемые инструменты и структурные элементы. Реализация новой возможности часто сводится к простому копированию/вставке; возможно, придется изменить одну-две настройки — и все будет готово.

Но это не отложенная загрузка!

Кто-то скажет: «Да, но почему вы называете эту загрузку отложенной? Счетчик все еще загружается сразу при загрузке страницы, просто это делается в отдельном запросе. Мы не ждем, пока его значение потребуется в приложении».

Хорошо. Сделаем отложенное действительно *отложенным*: запрос будет выдаваться только тогда, когда `span` станет видимым в результате прокрутки.

Для этого стоит вспомнить, как создавался пример с бесконечной прокруткой: мы воспользовались событием `revealed` в качестве триггера. Нужно просто сделать то же самое, верно? Выдавать запрос, когда элемент становится видимым?

Да, именно так. Снова для решения новой задачи в `htmx` можно объединить концепции разных UX-паттернов.

Листинг 93. Реализация полноценной отложенной загрузки

```
<span hx-get="/contacts/count" hx-trigger="revealed"> ❶
 <img id="spinner" class="htmx-indicator" src="/static/img/spinning-
circles.svg"/>
</span>
```

❶ Hx-trigger связывается с событием revealed.

Получилась действительно отложенная реализация, которая откладывает затратные вычисления до того момента, когда они будут в самом деле необходимы. Довольно полезный прием. И снова простое изменение одного атрибута демонстрирует гибкость как `htmx`, так и подхода гипермедиа.

Встроенное удаление

Реализуем еще один прием гипермедиа — паттерн «Встроенное удаление». В этом паттерне контакты можно удалять прямо из таблицы, не переходя к представлению редактирования отдельного контакта, чтобы получить доступ к кнопке удаления, добавленной в предыдущей главе.

Напомним, что в шаблоне `rows.html` в каждой строке таблицы уже присутствуют ссылки Edit (Редактирование) и View (Просмотр).

Листинг 94. Существующие действия со строками

```
<td>
  <a href="/contacts/{{ contact.id }}/edit">Edit</a>
  <a href="/contacts/{{ contact.id }}">View</a>
</td>
```

Теперь к ним также будет добавлена ссылка Delete (Удалить). И если подумать, эта ссылка должна делать то же самое, что и кнопка Delete Contact (Удалить контакт) из edit.html, не так ли? Необходимо выдать запрос HTTP `DELETE` к URL заданного контакта, а диалоговое окно подтверждения должно предотвратить непреднамеренное удаление контакта.

Ниже приведена разметка HTML для кнопки Delete Contact.

Листинг 95. Существующие действия со строками

```
<button hx-delete="/contacts/{{ contact.id }}"
    hx-push-url="true"
    hx-confirm="Are you sure you want to delete this contact?"
    hx-target="body">
  Delete Contact
</button>
```

Вы, наверное, уже догадались, что и в этом случае мы воспользуемся копированием/вставкой.

Обратим внимание, что в случае кнопки Delete Contact требуется заново отрендерить весь экран и обновить URL, так как приложение должно вернуться из представления редактирования контакта к представлению списка всех контактов. Однако в случае ссылки мы уже находимся в списке контактов, обновлять URL не нужно и атрибут `hx-push-url` можно опустить.

Код встроенной ссылки Delete выглядит так:

Листинг 96. Существующие действия со строками

```
<td>
  <a href="/contacts/{{ contact.id }}/edit">Edit</a>
  <a href="/contacts/{{ contact.id }}">View</a>
  <a href="#" hx-delete="/contacts/{{ contact.id }}"
    hx-confirm="Are you sure you want to delete this contact?"
    hx-target="body">Delete</a> ❶
</td>
```

❶ Почти полная копия кнопки Delete Contact.

Как видите, мы добавили новый якорный тег и назначили ему пустую цель (значение `#` в атрибуте `href`), чтобы сохранить для ссылки корректный стиль при наведении указателя мыши. Также мы скопировали атрибуты `hx-delete`, `hx-confirm` и `hx-target` из кнопки Delete Contact, но опустили атрибуты `hx-push-url`, чтобы избежать обновления URL в браузере.

Теперь встроенное удаление работает даже с диалоговым окном подтверждения. Пользователь может щелкнуть на ссылке Delete, и строка исчезнет из пользовательского интерфейса при повторном рендеринге страницы.

О СТИЛЕ

У такого способа удаления есть один побочный эффект: в правой части строки контакта выводится представление ссылок.

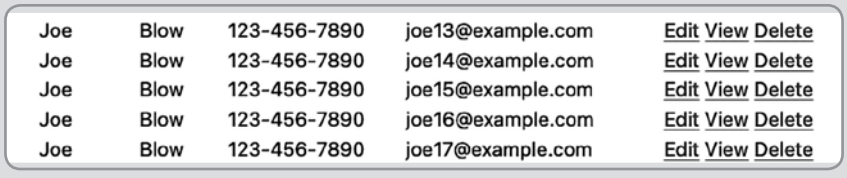


Рис. 8. Слишком много действий

Лучше, если бы в строке не выводились все возможные действия, а еще лучше – если бы действия появлялись только в том случае, если пользователь выразил интерес к этой конкретной строке. Мы еще вернемся к этой проблеме после того, как рассмотрим отношения между скриптами и гипермедиа-управляемым приложением в одной из следующих глав.

А пока просто смиритесь с этим далеко не идеальным интерфейсом. Позже мы его исправим.

Уточнение цели

Впрочем, есть и другие возможности для улучшения. Что, если вместо повторного рендеринга всей страницы просто удалить строку контакта? Пользователь все равно работает с этой строкой, так есть ли реальная необходимость заново рендерить всю страницу?

Вот что для этого нужно:

- Обновить ссылку, выбрав в качестве целевой текущую строку.
- Присвоить атрибуту `hx-swap` значение `outerHTML`, так как нам необходимо заменить (на самом деле удалить) всю строку.
- Обновить код на стороне сервера, чтобы он рендерил пустой контент при выдаче запроса `DELETE` от ссылки Delete вместо кнопки Delete Contact на странице редактирования контакта.

Обо всем по порядку: начнем с обновления цели ссылки Delete. Целевой должна быть строка, в которой находится ссылка, а не все тело. Здесь снова можно воспользоваться относительным позиционированием для выбора ближайшего тега `tr`, как это делалось при реализации паттернов «Загрузка по щелчку» и «Бесконечная прокрутка».

Листинг 97. Существующие действия со строками

```
<td>
  <a href="/contacts/{{ contact.id }}/edit">Edit</a>
  <a href="/contacts/{{ contact.id }}">View</a>
  <a href="#" hx-delete="/contacts/{{ contact.id }}"
```

```
    hx-swap="outerHTML"
    hx-confirm="Are you sure you want to delete this contact?"
    hx-target="closest tr">Delete</a> ❶
</td>
```

❶ Обновляет цель ближайшим внешним тегом tr (строка таблицы) ссылки.

Обновление кода на стороне сервера

Теперь необходимо обновить код на стороне сервера. Кнопка Delete Contact тоже должна работать, и в случае ее применения текущая логика верна. Следовательно, необходимо дифференцировать запросы `DELETE`, инициированные кнопкой, от запросов `DELETE`, инициированных якорной ссылкой.

Самое простое решение — добавить атрибут `id` к кнопке Delete Contact, чтобы проверить заголовок запроса HTTP `HX-Trigger` и определить, была ли кнопка удаления инициатором запроса. Это простое изменение существующей разметки HTML.

Листинг 98. Добавление атрибута id к кнопке Delete Contact

```
  <button id="delete-btn" ❶
      hx-delete="/contacts/{{ contact.id }}"
      hx-push-url="true"
      hx-confirm="Are you sure you want to delete this contact?"
      hx-target="body">
    Delete Contact
  </button>
```

❶ К кнопке добавляется атрибут id.

Назначая этой кнопке атрибут `id`, мы получаем возможность отличать кнопку удаления в шаблоне edit.html от ссылок удаления в шаблоне rows.html. Запрос, выданный кнопкой, выглядит примерно так:

```
DELETE http://example.org/contacts/42 HTTP/1.1
Accept: text/html,*/*
Host: example.org
...
HX-Trigger: delete-btn
...
```

Как видите, теперь запрос включает идентификатор кнопки. Это позволяет написать код, очень похожий на код активного поиска, с проверкой условия для заголовка `HX-Trigger`, чтобы определить, что нужно сделать. Если заголовок содержит значение `delete-btn`, мы понимаем, что запрос поступил от кнопки на странице редактирования, и можем сделать то же, что и сейчас: удалить контакт и выполнить перенаправление на страницу `/contacts`.

Если заголовок *не содержит* этого значения, можно просто удалить контакт и вернуть пустую строку. Эта пустая строка заменит цель, в данном случае строку таблицы для заданного контакта; в результате строка исключается из пользовательского интерфейса.

Проведем рефакторинг кода на стороне сервера.

Листинг 99. Обновление серверного кода для обработки двух разных паттернов удаления

```
@app.route("/contacts/<contact_id>", methods=["DELETE"])
def contacts_delete(contact_id=0):
  contact = Contact.find(contact_id)
  contact.delete()
  if request.headers.get('HX-Trigger') == 'delete-btn': ❶
    flash("Deleted Contact!")
    return redirect("/contacts", 303)
  else:
    return "" ❷
```

❶ Если запрос был отправлен кнопкой удаления на странице редактирования, продолжается выполнение предыдущей логики.

❷ А если нет, просто возвращается пустая строка, которая удалит строку таблицы.

Так выглядит обновленная реализация на стороне сервера: когда пользователь щелкает на ссылке Delete в строке контакта и подтверждает удаление, строка удаляется из пользовательского интерфейса. Это еще одна ситуация, в которой, изменив всего несколько строк простого кода, можно добиться радикального изменения поведения. Возможности гипермедиа в этом отношении огромны.

Модель замены в htmx

Все это неплохо, но есть еще одно улучшение, для реализации которого необходимо разобраться в модели замены контента в `htmx`: сделать так, чтобы удаляемая строка исчезала не сразу, а постепенно. Эффект постепенного скрытия визуализирует операцию удаления, предоставляя пользователю обратную связь об удалении.

Оказывается, с `htmx` реализовать его достаточно легко, но для этого необходимо хорошо понимать, как в `htmx` происходит замена контента.

Можно подумать, что `htmx` просто помещает новый контент в DOM, но на самом деле это не так. Вместо этого при добавлении контента в DOM выполняется последовательность действий.

- Когда принятый контент готов к подстановке в DOM, класс CSS `htmx-swapping` добавляется к целевому элементу.

- Затем происходит короткая задержка (вскоре мы обсудим, зачем она нужна).
- Далее класс `htmx-swapping` удаляется из цели и добавляется класс `htmx-settling`.
- Новый контент подставляется в DOM.
- Происходит еще одна короткая задержка.
- Наконец, класс `htmx-settling` удаляется из цели.

Это не полное описание механики замены (например, `htmx-settling` — более сложная тема, которая будет рассмотрена в одной из следующих глав), но пока его достаточно.

В ходе процесса возникают небольшие задержки, как правило, порядка нескольких миллисекунд. Почему? Оказывается, эти задержки нужны для выполнения *переходов CSS*.

ПЕРЕХОДЫ CSS

Переходы CSS – технология, позволяющая анимировать переходы между стилями. Например, если вы увеличили высоту элемента с 10 до 20 пикселей, переход CSS позволяет выполнить плавную анимацию роста элемента до новой величины. Такие анимации забавны, с ними приложение выглядит привлекательнее для пользователей, и это отличный способ добавить в него глянца.

К сожалению, с переходами CSS трудно работать в простом HTML: обычно приходится использовать JavaScript, добавляя или удаляя классы для их срабатывания. Из-за этого модель замены `htmx` сложнее, чем можно ожидать изначально. Заменяя классы и добавляя короткие задержки, можно обращаться к переходам CSS исключительно из HTML, так что вам не придется писать код JavaScript!

Использование htmx-swapping

Итак, вернемся немного назад и присмотримся к механике встроенного удаления: мы щелкаем на ссылке, дополненной `htmx`, которая удаляет контакт, а затем подставляет пустой контент на место строки. Мы знаем, что перед удалением элемента `tr` к нему будет добавлен класс `htmx-swapping`. Можно воспользоваться этим для написания перехода CSS, который понижает уровень непрозрачности строки до 0. CSS будет выглядеть так:

Листинг 100. Добавление перехода постепенного исчезновения

```
tr.htmx-swapping { ❶
 opacity: 0; ❷
 transition: opacity 1s ease-out; ❸
}
```

❶ Стиль должен применяться к элементам tr с классом htmx-swapping.
❷ Уровень непрозрачности равен 0, так что элемент становится невидимым.
❸ Уровень непрозрачности снижается до 0 за время в 1 секунду с использованием функции ease-out.

CSS не является темой этой книги, так что мы не будем подробно разбираться в переходах CSS. Будем надеяться, что приведенный код вам понятен, даже если вы впервые видите переход CSS.

Итак, подумайте, что это означает с точки зрения модели замены `htmx`: когда библиотека `htmx` получает контент, который должен быть подставлен в строку, она назначает строке класс `htmx-swapping` и немного выжидает. Благодаря этому может выполниться переход к нулевому уровню непрозрачности, в результате чего строка постепенно исчезнет. Затем будет подставлен новый (пустой) контент, что фактически приведет к удалению строки.

Звучит неплохо, и задача почти решена. Остается разобраться еще кое с чем: «задержка замены» по умолчанию для `htmx` очень мала, она составляет всего несколько миллисекунд. Во многих случаях это оправданно: перед появлением нового контента в DOM большой задержки быть не должно. Но в нашем случае мы хотим предоставить анимации CSS время для завершения перед заменой, и задержку следует увеличить до секунды.

К счастью, в `htmx` предусмотрена аннотация `hx-swap`, которая позволяет назначить задержку замены. После типа замены добавляется модификатор `swap:` с временны́м интервалом. Он сообщает `htmx`, какое время следует выждать перед заменой. Обновим разметку HTML, чтобы замена для действия удаления выполнялась после задержки в 1 секунду.

Листинг 101. Существующие действия со строками

```
<td>
  <a href="/contacts/{{ contact.id }}/edit">Edit</a>
  <a href="/contacts/{{ contact.id }}">View</a>
  <a href="#" hx-delete="/contacts/{{ contact.id }}"
    hx-swap="outerHTML swap:1s" ❶
    hx-confirm="Are you sure you want to delete this contact?"
    hx-target="closest tr">Delete</a>
</td>
```

❶ Задержка замены изменяет продолжительность паузы, которую делает htmx перед подстановкой нового контента.

С таким изменением существующая строка будет оставаться в DOM в течение дополнительной секунды, при этом ей назначается класс `htmx-swapping`. Это даст строке время выполнить переход к нулевому уровню прозрачности, с созданием нужного эффекта постепенного исчезновения.

Теперь, когда пользователь щелкает по ссылке Delete и подтверждает удаление, строка постепенно становится невидимой, и по достижении уровня непрозрачности 0 она будет удалена. И все это делается декларативным, гипермедиа-ориентированным способом, без участия JavaScript. (Очевидно, библиотека `htmx` написана на JavaScript, но вы знаете, о чем мы: вам не придется писать код JavaScript для реализации этой возможности.)

Групповое удаление

Последняя возможность, которую мы реализуем в этой главе, — «групповое удаление». Текущий механизм удаления пользователей удобен, но только до тех пор, пока пользователю не понадобится удалить пять или десять контактов сразу. Для реализации группового удаления необходимо добавить режим удаления строк, при котором пользователь устанавливает флажки в нескольких строках, а затем удаляет их все сразу щелчком кнопки Delete Selected Contacts (Удалить выбранные контакты).

Чтобы реализовать эту возможность, необходимо добавить флажок в каждую строку в шаблоне rows.html. Этому полю ввода будет присвоено имя `selected_contact_ids`, а его значением будет идентификатор контакта для текущей строки.

Вот как выглядит обновленный код rows.html.

Листинг 102. Добавление флажка в каждую строку

```
{% for contact in contacts %}
<tr>
 <td><input type="checkbox" name="selected_contact_ids" value="{{ contact.id
}}"></td> ❶
 <td>{{ contact.first }}</td>
 ...
</tr>
{% endfor %}
```

❶ Новая ячейка с полем ввода checkbox, значением которой является идентификатор текущего контакта.

Также необходимо добавить пустой столбец в заголовок таблицы для столбца с флажком. Когда это будет сделано, мы получим серию флажков, по одному для каждой строки таблицы, — несомненно, этот паттерн знаком вам по многим веб-сайтам.

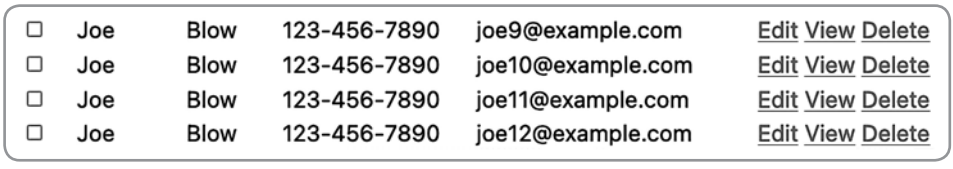


Рис. 9. Флажки для строк таблицы с контактами

Если вы не знаете или забыли, как работают флажки (чекбоксы) в HTML, напоминаем: флажок отправляет свое значение, связанное с именем поля ввода, в том и только в том случае, если он установлен. Если, например, вы установите флажки контактов с идентификаторами 3, 7 и 9, то серверу будут отправлены только эти три значения. Так как все флажки в данном случае имеют одинаковые имена `selected_contact_ids`, все три значения будут отправлены с именем `selected_contact_ids`.

Кнопка Delete Selected Contacts (Удалить выбранные контакты)

На следующем шаге под таблицей добавляется кнопка, которая удалит все выделенные контакты. Необходимо, чтобы эта кнопка, как и ссылки удаления в каждой строке, выдавала запрос HTTP `DELETE`, но вместо URL для конкретного контакта, как это делают встроенные ссылки удаления и кнопка удаления на странице редактирования, запрос `DELETE` должен выдаваться к URL `/contacts`.

Как и с другими элементами удаления, необходимо убедиться, что пользователь действительно хочет удалить контакты; для этого целевым будет выбрано тело страницы, так как мы будем заново рендерить всю таблицу.

Код кнопки выглядит так:

Листинг 103. Кнопка удаления выделенных контактов

```
<button hx-delete="/contacts" ❶
    hx-confirm="Are you sure you want to delete these contacts?" ❷
    hx-target="body"> ❸
  Delete Selected Contacts
</button>
```

❶ Выдает запрос DELETE к /contacts.

❷ Проверяет, что пользователь действительно хочет удалить выделенные контакты («Вы уверены, что хотите удалить эти контакты?»).

❸ Назначает целью body.

Ничего сложного. Только один вопрос: как включить значения всех установленных флажков в запрос? В текущей версии это просто автономная кнопка, которой неизвестно, что она должна включать в выдаваемый запрос `DELETE` какую-либо дополнительную информацию.

К счастью, в `htmx` есть несколько способов добавления значений полей ввода в запросы.

Один из способов основан на применении атрибута `hx-include`, позволяющего использовать селектор CSS для указания элементов, включаемых в запрос. Такое решение сработает, но мы пойдем другим путем, который немного проще.

Если элемент является дочерним для элемента формы и выдает запрос, отличный от `GET`, `htmx` по умолчанию включает значения всех полей ввода этой формы. В подобных ситуациях с выполнением групповых операций с таблицей в тег формы обычно включается вся таблица; это позволяет легко добавить кнопки, работающие с выделенными элементами.

Заключим таблицу в тег формы (не забудьте включить кнопку вместе с таблицей).

Листинг 104. Кнопка удаления выделенных контактов

```
<form> ❶
  <table>
   ... omitted
  </table>
  <button hx-delete="/contacts"
      hx-confirm="Are you sure you want to delete these contacts?"
      hx-target="body">
    Delete Selected Contacts
  </button>
</form> ❷
```

❶ Тег form включает всю таблицу.
❷ Тег form также включает кнопку.

Когда кнопка выдает запрос `DELETE`, обновленная версия добавит все идентификаторы контактов, выбранных пользователем, в переменную запроса `selected_contact_ids`.

Серверный код для удаления выделенных элементов

Реализация на стороне сервера напоминает исходный серверный код удаления контактов. Собственно, вы снова можете скопировать код и немного его изменить.

- Измените URL на `/contacts`.
- Обработчик должен получить *все* отправленные идентификаторы под именем `selected_contact_ids` и перебрать их, удаляя каждый контакт по отдельности.

И это все! Код на стороне сервера будет выглядеть так:

Листинг 105. Кнопка удаления выделенных контактов

```
@app.route("/contacts/", methods=["DELETE"]) ❶
def contacts_delete_all():
  contact_ids = list(map(int,
request.form.getlist("selected_contact_ids"))) ❷
  for contact_id in contact_ids: ❸
    contact = Contact.find(contact_id)
    contact.delete() ❹
  flash("Deleted Contacts!") ❺
  contacts_set = Contact.all()
  return render_template("index.html", contacts=contacts_set)
```

❶ Обрабатывает запрос DELETE к пути /contacts/.

❷ Преобразует значения selected_contact_ids, отправленные серверу, из списка строк в список целых чисел.

❸ Перебирает все идентификаторы.

❹ Удаляет контакт, связанный с каждым идентификатором.

❺ Остальной код остается таким же, что и в исходном обработчике удаления: он создает флеш-сообщение и рендерит шаблон index.html.

Итак, мы взяли исходную логику удаления и слегка изменили ее, чтобы она работала с массивом идентификаторов вместо отдельного идентификатора.

Возможно, вы заметили еще одно изменение: мы избавились от перенаправления, присутствовавшего в исходном коде удаления. Это было сделано, потому что мы уже находимся на странице, которую необходимо отрендерить заново, так что незачем выполнять перенаправление и замену URL другим значением. Можно просто заново отрендерить страницу, тогда новый список контактов (без удаленных) также будет отрендерен заново.

Все готово: в приложении реализована возможность группового удаления. И снова объем кода невелик, а вся функциональность реализована исключительно обменом гипермедиа с сервером в традиционной манере веб-среды, в соответствии с принципами REST.

Заметки об HTML: доступность по умолчанию?

При реализации элементов управления, не встроенных в HTML, могут возникнуть проблемы с их доступностью.

Ранее, в главе 1, был рассмотрен пример элемента `<div>`, который работал как кнопка. Возьмем другой пример: что, если создать элемент, который выглядит как набор вкладок, но для его построения используются радиокнопки и приемы CSS? Это полезная техника, которую иногда используют веб-разработчики.

Проблема в том, что у вкладок есть другие требования, кроме щелчков для изменения контента. У ваших импровизированных вкладок может отсутствовать какая-то функциональность, что вызовет путаницу и раздражение у пользователя, а также приведет к нежелательному поведению. Выдержка из руководства ARIA Authoring Practices Guide[1], касающаяся вкладок:

- Взаимодействие с клавиатурой
  - Поддерживается ли переход между вкладками клавишей Tab?
- Роли, состояния и свойства ARIA
  - [Элемент, содержащий вкладки] имеет роль `tablist`.
  - Каждая [вкладка] имеет роль `tab` [...].
  - Каждый элемент, содержащий панель контента для вкладки, имеет роль `tabpanel`.
  - Каждая [вкладка] имеет свойство `aria-controls`, содержащее ссылку на связанный с ней элемент `tabpanel`.
  - У активного элемента `tab` состоянию `aria-selected` присваивается `true`, а у остальных элементов вкладок оно установлено в `false`.
  - Каждый элемент с ролью `tabpanel` содержит свойство `aria-labelledby`, ссылающееся на связанный элемент `tab`.

Чтобы создать собственный набор вкладок, соответствующий всем этим требованиям, придется написать много кода. Некоторые атрибуты ARIA можно включить прямо в HTML, но это однообразная работа, а некоторые из них (такие, как `aria-selected`) должны задаваться из JavaScript, так как они являются динамическими. Клавиатурные взаимодействия тоже повышают риск ошибок.

Написать собственную реализацию набора вкладок можно и даже не так сложно. Тем не менее трудно рассчитывать на то, что новая реализация будет работать у всех пользователей во всех средах, так как у большинства разработчиков ресурсы для тестирования ограниченны.

Старайтесь пользоваться проверенными библиотеками для UI-взаимодействий. Если ситуация требует индивидуального решения, тщательно протестируйте клавиатурные взаимодействия и доступность. Проведите ручное тестирование. Проведите автоматизированное тестирование. Тестируйте с экранными дикто-

[1] https://www.w3.org/WAI/ARIA/apg/patterns/tabs/

рами, тестируйте с клавиатурой, тестируйте с разными браузерами и оборудованием, запускайте линтеры (при написании кода и/или непрерывной интеграции). Тестирование очень важно для обеспечения удобочитаемости кода человеком или машиной или оптимального веса страницы.

Ответьте также на вопрос: обязательно ли использовать для представления информации набор вкладок? Иногда ответ будет «да», но если ответ «нет» — набор раскрывающихся детализаций может выполнять очень похожие функции.

```
<details><summary>Disclosure 1</summary>
 Disclosure 1 contents
</details>
<details><summary>Disclosure 2</summary>
 Disclosure 2 contents
</details>
```

Сознательное снижение качества UX только для того, чтобы избежать использования JavaScript, — плохая практика разработки. Но иногда можно добиться равного (или лучшего!) качества UX при более простой и надежной реализации.

UI ДИНАМИЧЕСКОЙ АРХИВАЦИИ

Приложение Contact.app прошло долгий путь от традиционной реализации в стиле Web 1.0: в него был добавлен активный поиск, групповое удаление, анимации и ряд других возможностей. Мы добились уровня интерактивности, для которого многие веб-разработчики применяют фреймворки JavaScript SPA, но сделали это исключительно средствами гипермедиа на основе `htmx`.

Посмотрим, как добавить в Contact.app последнюю важную функцию: загрузку архива всех контактов.

С точки зрения гипермедиа загрузку файлов реализовать не очень сложно: по заголовку ответа HTTP `Content-Disposition` можно указать браузеру загрузить и сохранить файл локально на компьютере.

Но чтобы задача была более интересной, добавим условие: экспорт данных должен занимать некоторое время: 5–10 секунд или чуть больше.

Таким образом, при реализации загрузки как «обычного» запроса HTTP, инициируемого по ссылке или кнопке, пользователю придется какое-то время ожидать, не имея визуальной обратной связи. Он не будет понимать, происходит ли загрузка. Может быть, он даже выйдет из себя и снова щелкнет на элементе управления загрузкой, отправив *второй* запрос на архивацию. Нехорошо.

Это классическая проблема веб-разработки. Когда вы сталкиваетесь с подобными процессами, которые могут занимать какое-то время, возможны два варианта:

- когда пользователь инициирует действие, блокировать приложение до завершения действия, а затем ответить результатом;
- начать действие и немедленно вернуть управление с выводом элемента UI, показывающего, что операция продолжает выполняться.

Безусловно, блокировка с ожиданием завершения действия — более простой вариант, но он может произвести плохое впечатление на пользователя, особенно если на завершение действия требуется время. Если вы когда-нибудь щел-

кали на элементах приложений в стиле Web 1.0, а затем приходилось целую вечность (как вам казалось) дожидаться, когда в приложении что-нибудь произойдет, — вы уже знаете эффект.

Второй вариант с асинхронным запуском действия (предположим, созданием потока или передачей его системе выполнения задач) намного предпочтительнее с точки зрения опыта взаимодействия с пользователем: сервер может ответить немедленно, и пользователю не придется долго ждать и думать, что же происходит.

Вопрос в том, *чем* отвечать? Скорее всего, задание еще не завершится, так что выдать ссылку на результат не получится.

Мы видели несколько «простых» решений для подобных ситуаций в разных веб-приложениях.

- Сообщить пользователю, что процесс начат, и после его завершения отправить по электронной почте ссылку на результаты.
- Сообщить пользователю, что процесс начат, и порекомендовать ему вручную обновить страницу, чтобы получить информацию о текущем состоянии процесса.
- Сообщить пользователю, что процесс начат, и автоматически обновлять страницу каждые несколько секунд при помощи кода JavaScript.

Все эти решения работают, но ни одно из них нельзя назвать хорошим с точки зрения взаимодействия с пользователем.

Действительно хороший вариант — вывести элемент наподобие того, что вы видите, например, при загрузке большого файла в браузере: индикатор прогресса показывает, на какой стадии находится процесс, а когда процесс завершится, выводится ссылка для просмотра результата.

Может показаться, что эту задачу не решить средствами гипермедиа и, откровенно говоря, нам придется выжать из `htmx` максимум, чтобы решение заработало, но в итоге объем кода будет *не очень* большим и мы сможем реализовать нужный опыт взаимодействия для функции архивации.

Требования к UI

Прежде чем переходить к реализации, представим в общих чертах, как будет выглядеть новый интерфейс: в приложении должна присутствовать кнопка с ярлыком Download Contact Archive (Загрузить архив контактов). По щелчку пользователя эта кнопка должна сменяться пользовательским интерфейсом, показывающим прогресс операции архивации, в идеале с индикатором прогресса. По мере выполнения архивации индикатор должен перемещаться к точ-

ке завершения. По завершении задания архивации должна появляться ссылка для загрузки архива контактов.

Для реализации процесса архивации воспользуемся классом Python `Archiver`, который предоставляет всю необходимую функциональность. Как и в случае с классом `Contact`, мы не будем углубляться в детали, потому что эта тема выходит за рамки книги. Пока достаточно знать, что класс предоставляет все серверное поведение, необходимое для запуска процесса архивации контактов и получения результата по его завершении.

Класс `Archiver` предоставляет следующие методы:

- `status()` — строка с описанием статуса загрузки (`Waiting`, `Running` или `Complete`);
- `progress()` — число от 0 до 1, показывающее, какая часть задания архивации выполнена;
- `run()` — запускает новое задание архивации (если текущий статус равен `Waiting`);
- `reset()` — отменяет текущее задание архивации, если оно есть, и возвращается к состоянию `Waiting`;
- `archive_file()` — путь к файлу архива, созданному на сервере, для отправки клиенту;
- `get()` — метод класса, который позволяет получить экземпляр `Archiver` для текущего пользователя.

Как видите, API довольно прост.

Единственный неочевидный аспект заключается в том, что метод `run()` — *неблокирующий*. Это означает, что он не создает файл с архивом *немедленно*, а запускает фоновое задание (в отдельном потоке) для выполнения архивации. Это может быть сложно, если вы не привыкли к многопоточности в коде: возможно, вы ожидаете, что метод `run()` будет «блокирующим», то есть выполнит весь экспорт данных и вернет управление только после завершения операции. Но в таком случае невозможно запустить процесс архивации и сразу выполнить рендеринг индикатора прогресса.

Начало работы

У нас есть все необходимое для начала реализации UI: продуманная схема интерфейса и логика предметной области, обеспечивающая его поддержку.

Итак, для начала отметим, что UI в целом самодостаточен: нам нужно заменить кнопку индикатором прогресса загрузки, а затем индикатор прогресса — ссылкой для загрузки результатов завершенного процесса архивации.

Тот факт, что пользовательский интерфейс архивации будет содержаться в конкретной части UI, подсказывает, что для работы с ним стоит создать новый шаблон. Назовем этот шаблон archive_ui.html.

Также обратите внимание, что весь пользовательский интерфейс загрузки должен заменяться несколько раз.

- В начале загрузки кнопка заменяется индикатором прогресса.
- В процессе архивации необходимо заменять/обновлять индикатор прогресса.
- После завершения процесса архивации индикатор прогресса заменяется ссылкой на загрузку.

Чтобы UI обновлялся подобным образом, необходимо верно определить цель для обновлений. Упакуем весь UI в тег `div`, который затем будет целевым для всех операций.

Начало шаблона нового пользовательского интерфейса архивации выглядит так:

Листинг 106. Исходный шаблон архивации UI

```
<div id="archive-ui"
  hx-target="this" ❶
  hx-swap="outerHTML"> ❷
</div>
```

❶ Этот тег div будет целевым для всех содержащихся в нем элементов.
❷ Каждый раз весь тег div заменяется при помощи outerHTML.

Затем добавим к `div` кнопку Download Contact Archive (Загрузить архив контактов), которая будет запускать процесс архивации с последующей загрузкой. Для запуска процесса архивации будет использоваться запрос `POST` к пути `/contacts/archive`.

Листинг 107. Добавление кнопки архивации

```
<div id="archive-ui" hx-target="this" hx-swap="outerHTML">
 <button hx-post="/contacts/archive"> ❶
   Download Contact Archive
 </button>
</div>
```

❶ Эта кнопка выдает запрос POST к /contacts/archive.

Наконец, новый шаблон включается в главный шаблон index.html над таблицей контактов.

Листинг 108. Исходный шаблон пользовательского интерфейса архивации

```
{% block content %}

  {% include 'archive_ui.html' %} ❶

  <form action="/contacts" method="get" class="tool-bar">
```

❶ Этот шаблон будет включен в главный шаблон.

Когда это будет сделано, в веб-приложении появится кнопка для загрузки. Так как у внешнего тега `div` установлен атрибут `hx-target="this"`, кнопка унаследует цель и заменит внешний `div` разметкой HTML, возвращенной по запросу `POST` к `/contacts/archive`.

Добавление конечной точки архивации

Следующим шагом станет обработка запроса `POST`, выдаваемого кнопкой. Нам необходимо получить экземпляр `Archiver` для текущего пользователя и вызвать для него метод `run()`. Вызов метода запускает процесс архивации, после чего рендерится новый контент, указывающий, что процесс продолжает выполняться.

Чтобы это реализовать, повторно воспользуемся шаблоном `archive_ui` для обработки рендеринга UI архивации в обоих состояниях: Waiting и Running. (Обработка состояния завершения Complete будет рассмотрена немного позже.)

Это очень популярный паттерн: вы помещаете все потенциальные варианты UI для фрагмента пользовательского интерфейса в один шаблон, а затем рендерите нужный вариант в соответствии с условием. Когда все варианты хранятся в одном файле, другим разработчикам (или нам самим, если мы вернемся к коду через какое-то время!) будет проще понять, как UI работает на стороне клиента.

Так как мы собираемся рендерить разные пользовательские интерфейсы на основании состояния архиватора, нужно будет передать архиватор шаблону в параметре. Еще раз: мы будем вызывать `run()` для архиватора в контроллере, а затем передавать архиватор шаблону, чтобы тот отрендерил вариант UI для текущего статуса процесса архивации.

Код выглядит так:

Листинг 109. Код на стороне сервера для запуска процесса архивации

```
@app.route("/contacts/archive", methods=["POST"]) ❶
def start_archive():
  archiver = Archiver.get() ❷
  archiver.run() ❸
  return render_template("archive_ui.html", archiver=archiver) ❹
```

❶ Обрабатывает запрос POST к /contacts/archive.
❷ Получает Archiver.
❸ Вызывает неблокирующий метод run().
❹ Рендерит шаблон archive_ui.html с передачей архиватора.

Рендеринг UI прогресса операции по условию

Теперь перейдем к обновлению UI архивации. Для этого настроим archive_ui.html для рендеринга разного контента в зависимости от состояния процесса архивации.

Вспомните, что архиватор содержит метод `status()`. Когда мы передаем архиватор шаблону в переменной, мы можем проверить статус процесса архивации при помощи метода `status()`.

Если архиватор имеет статус `Waiting`, рендериться должна кнопка Download Contact Archive (Загрузить архив контактов). В случае статуса `Running` должно рендериться сообщение о выполнении операции. Внесем соответствующие изменения в код шаблона.

Листинг 110. Добавление рендеринга по условию

```
<div id="archive-ui" hx-target="this" hx-swap="outerHTML">
  {% if archiver.status() == "Waiting" %} ❶
    <button hx-post="/contacts/archive">
      Download Contact Archive
    </button>
  {% elif archiver.status() == "Running" %} ❷
   Running... ❸
  {% end %}
</div>
```

❶ Кнопка архивации рендерится только при статусе Waiting.
❷ Рендерит другой контент при статусе Running.
❸ Пока что рендерится текстовое сообщение.

Хорошо, мы реализовали условную логику в шаблоне, а также логику на стороне сервера для поддержки запуска процесса архивации. Индикатор прогресса еще не готов, но мы его непременно сделаем! Посмотрим, как работает текущая версия приложения, и обновим его главную страницу...

Листинг 111. Что-то пошло не так

```
UndefinedError
jinja2.exceptions.UndefinedError: 'archiver' is undefined
```

Ой!

Сразу выдается сообщение об ошибке. Почему? Дело в том, что мы включаем archive_ui.html в шаблон index.html, но теперь шаблон archive_ui.html ожидает, что ему будет передан архиватор для рендеринга верного варианта UI по условию.

Проблема решается просто: достаточно передавать архиватор при рендеринге шаблона index.html.

Листинг 112. Включение архиватора при рендеринге index.html

```
@app.route("/contacts")
def contacts():
  search = request.args.get("q")
  if search is not None:
    contacts_set = Contact.search(search)
    if request.headers.get('HX-Trigger') == 'search':
      return render_template("rows.html", contacts=contacts_set)
  else:
    contacts_set = Contact.all()
  return render_template("index.html", contacts=contacts_set,
archiver=Archiver.get()) ❶
```

❶ Архиватор передается главному шаблону.

Когда это будет сделано, можно загрузить страницу. Как и следовало ожидать, на ней присутствует кнопка Download Contact Archive.

Если щелкнуть на ней, кнопка сменится сообщением «Running...», и в консоли разработчика на стороне сервера будет видно, что задание действительно запущено.

Опрос

Конечно, это шаг вперед, но такой индикатор прогресса далеко не идеален: это просто статический текст, который оповещает пользователя о том, что процесс выполняется.

Лучше, если контент будет обновляться по мере выполнения архивации, а в идеале будет отображаться индикатор прогресса. Как это сделать в `htmx` стандартными средствами гипермедиа?

Прием, которым мы воспользуемся, называется «опросом» (polling); приложение выдает запрос через определенный интервал и обновляет пользовательский интерфейс в зависимости от нового состояния сервера.

`Htmx` поддерживает две разновидности опроса. Первый — «опрос с фиксированной частотой» — использует специальный синтаксис `hx-trigger` для обозначения того, что опрос должен проводиться с фиксированными интервалами.

ОПРОС? СЕРЬЕЗНО?

У опроса скверная репутация, и это не самый элегантный прием в мире: в наши дни разработчики стараются использовать в подобных ситуациях более современные средства, такие как WebSockets или SSE (Server Sent Events).

Но кто бы что ни говорил, опрос *работает*, и трудно придумать что-то проще. Нужно действовать осторожно, чтобы не перегрузить систему периодическими запросами, но при должной внимательности можно создать надежный, пассивно обновляемый компонент UI.

Пример:

Листинг 113. Опрос с фиксированными интервалами

```
<div hx-get="/messages" hx-trigger="every 3s"> ❶
</div>
```

❶ Инициирует запрос GET к /messages через каждые три секунды.

Такое решение хорошо работает, если опрос должен вестись бесконечно, например для постоянной проверки новых сообщений, которые должны выводиться для пользователя. Однако опрос с фиксированной частотой неидеален, если его необходимо прекратить по окончании определенного процесса: при фиксированной частоте опрос продолжается до тех пор, пока элемент, с которым он связан, не будет удален из DOM.

В нашем случае присутствует определенный конечный процесс. А значит, будет лучше воспользоваться вторым видом опроса, известным как «опрос при загрузке» (load polling). В этом случае используется тот факт, что `htmx` инициирует событие `load` при загрузке контента в DOM. Можно создать триггер для этого события `load` и добавить короткую задержку, чтобы запрос не инициировался немедленно.

В таком случае можно рендерить `hx-trigger` с условием при каждом запросе: когда запрос завершится, мы просто не добавляем триггер `load`, и опрос при загрузке останавливается. Тем самым обеспечивается простой и удобный механизм опроса до завершения определенного процесса.

Использование опроса для обновления UI архивации

Воспользуемся опросом при загрузке для обновления пользовательского интерфейса по мере выполнения архивации. Для наглядного представления прогресса будет применяться индикатор на основе CSS. В нем используется метод

`progress()`, который возвращает число от 0 до 1, показывающее, насколько процесс архивации близок к завершению.

Для этого будет использоваться следующий фрагмент HTML.

Листинг 114. Индикатор прогресса на основе CSS

```
<div class="progress">
  <div class="progress-bar"
    style="width:{{ archiver.progress() * 100 }}%"></div> ❶
</div>
```

❶ Ширина внутреннего элемента соответствует степени прогресса.

Индикатор прогресса на основе CSS состоит из двух компонентов: внешнего тега `div`, предоставляющего основу для индикатора прогресса, и внутреннего тега `div`, который, собственно, и является индикатором. Ширина внутреннего индикатора задается равной проценту завершения (обратите внимание: для получения процента необходимо умножить результат `progress()` на 100), и в итоге индикатор будет иметь подходящую ширину внутри родительского тега `div`.

А КАК ЖЕ ЭЛЕМЕНТ <PROGRESS>?

Использование тега `div` вместо абсолютно типичного тега HTML5 `progress`, который (сюрприз!) был создан специально для обозначения прогресса, чревато риском оказаться в «каше из div».

Мы решили не использовать элемент `progress` в данном случае, потому что индикатор должен обновляться плавно, а для этого придется привлекать средства CSS, недоступные для элемента `progress`. Это печально, но иногда приходится играть теми картами, которые вам достались.

Впрочем, мы будем использовать роли индикатора прогресса, чтобы индикатор прогресса на основе `div` хорошо сочетался с технологиями доступности.

Обновим индикатор прогресса подходящими ролями и значениями ARIA.

Листинг 115. Индикатор прогресса на основе CSS

```
<div class="progress">
  <div class="progress-bar"
    role="progressbar" ❶
    aria-valuenow="{{ archiver.progress() * 100}}}" ❷
    style="width:{{ archiver.progress() * 100 }}%"></div> ❶
</div>
```

❶ Этот элемент служит индикатором прогресса.
❷ Прогресс отображается как процент выполненного объема работы архиватора, где 100 — полное завершение.

Наконец, для полноты приведем код CSS, который будет использоваться для индикатора прогресса.

Листинг 116. CSS индикатора прогресса

```
.progress {
  height: 20px;
  margin-bottom: 20px;
  overflow: hidden;
  background-color: #f5f5f5;
  border-radius: 4px;
  box-shadow: inset 0 1px 2px rgba(0,0,0,.1);
}
.progress-bar {
  float: left;
  width: 0%;
  height: 100%;
  font-size: 12px;
  line-height: 20px;
  color: #fff;
  text-align: center;
  background-color: #337ab7;
  box-shadow: inset 0 -1px 0 rgba(0,0,0,.15);
  transition: width .6s ease;
}
```

А вот как будет выглядеть индикатор на странице:

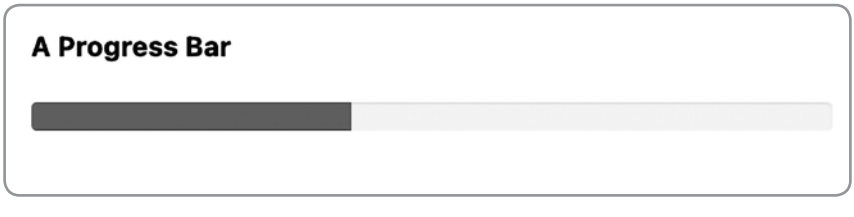


Рис. 10. Индикатор прогресса на основе CSS

Добавление UI для индикатора прогресса

Добавим в шаблон archive_ui.html код индикатора прогресса для случая выполнения архивации и обновим копию текстом «Creating Archive...» (Выполняется создание архива...).

Листинг 117. Добавление индикатора прогресса

```
<div id="archive-ui" hx-target="this" hx-swap="outerHTML">
  {% if archiver.status() == "Waiting" %}
    <button hx-post="/contacts/archive">
      Download Contact Archive
    </button>
  {% elif archiver.status() == "Running" %}
    <div>
      Creating Archive...
      <div class="progress" > ❶
        <div class="progress-bar" role="progressbar"
          aria-valuenow="{{ archiver.progress() * 100}}"
          style="width:{{ archiver.progress() * 100 }}%"></div>
      </div>
    </div>
  {% endif %}
</div>
```

❶ Новый индикатор прогресса.

Теперь по щелчку кнопки Download Contact Archive выводится индикатор прогресса. Однако он не обновляется, потому что мы еще не реализовали опрос при загрузке: индикатор просто остается на нуле. Чтобы индикатор обновлялся динамически, необходимо реализовать опрос при загрузке с использованием `hx-trigger`. Код можно добавить практически в любой элемент внутри условного блока для выполняемого архиватора; мы добавим его в тег `div`, внешний по отношению к тексту «Выполняется создание архива...» и индикатору прогресса.

Чтобы запустить опрос, выдадим запрос HTTP `GET` к тому же пути, что и `POST`: `/contacts/archive`.

Листинг 118. Реализация опроса при загрузке

```
<div id="archive-ui" hx-target="this" hx-swap="outerHTML">
  {% if archiver.status() == "Waiting" %}
    <button hx-post="/contacts/archive">
      Download Contact Archive
    </button>
  {% elif archiver.status() == "Running" %}
    <div hx-get="/contacts/archive" hx-trigger="load delay:500ms"> ❶
      Creating Archive...
      <div class="progress" >
        <div class="progress-bar" role="progressbar"
          aria-valuenow="{{ archiver.progress() * 100}}"
          style="width:{{ archiver.progress() * 100 }}%"></div>
      </div>
    </div>
  {% endif %}
</div>
```

❶ Выдает запрос GET к /contacts/archive через 500 миллисекунд после загрузки контента.

Когда запрос `GET` выдается к пути `/contacts/archive`, он заменяет `div` с идентификатором `archive-ui`, а не только самого себя. Атрибут `hx-target` для `div` с идентификатором `archive-ui` *наследуется* всеми дочерними элементами внутри этого тега `div`, так что для всех дочерних элементов целевым является внешний тег `div` в файле archive_ui.html.

Теперь необходимо обработать запрос `GET` к `/contacts/archive` на сервере. К счастью, это просто: все, что для этого нужно, — заново отрендерить archive_ui.html с архиватором.

Листинг 119. Обработка обновлений прогресса

```
@app.route("/contacts/archive", methods=["GET"]) ❶
def archive_status():
  archiver = Archiver.get()
  return render_template("archive_ui.html", archiver=archiver) ❷
```

❶ Обрабатывает запрос GET к пути /contacts/archive.
❷ Заново рендерит шаблон archive_ui.html.

Как обычно и бывает в приложениях гипермедиа, код легко читается и несложен.

Если теперь щелкнуть по кнопке Download Contact Archive, индикатор прогресса будет обновляться через каждые 500 миллисекунд. По мере того как результат вызова `archiver.progress()` последовательно увеличивается от 0 до 1, индикатор прогресса движется на экране. Впечатляет!

Загрузка результата

Осталось обработать еще одно состояние, в котором `archiver.status()` содержит Complete (Завершено), а следовательно, существует архив с данными JSON, готовый к загрузке. Когда архиватор завершит работу, вы сможете получить локальный файл JSON на сервере от архиватора вызовом `archive_file()`.

Добавим в выражение `if` еще один случай для обработки состояния Complete. По завершении задания архивации рендерится ссылка на новый путь `/contacts/archive/file`, направляющая в ответ архивный файл JSON. Новый код выглядит так:

Листинг 120. Рендеринг ссылки на загрузку по завершении архивации

```
<div id="archive-ui" hx-target="this" hx-swap="outerHTML">
  {% if archiver.status() == "Waiting" %}
    <button hx-post="/contacts/archive">
```

```
      Download Contact Archive
    </button>
  {% elif archiver.status() == "Running" %}
    <div hx-get="/contacts/archive" hx-trigger="load delay:500ms">
      Creating Archive...
      <div class="progress" >
        <div class="progress-bar" role="progressbar"
          aria-valuenow="{{ archiver.progress() * 100}}"
          style="width:{{ archiver.progress() * 100 }}%"></div>
      </div>
    </div>
  {% elif archiver.status() == "Complete" %} ❶
    <a hx-boost="false" href="/contacts/archive/file">Archive Ready!
Click here to download. &downarrow;</a> ❷
  {% endif %}
</div>
```

❶ Для статуса Complete рендерит ссылку на загрузку.
❷ Ссылка выдает запрос GET к /contacts/archive/file.

Обратите внимание: у ссылки атрибуту `hx-boost` присваивается `false`. Это делается для того, чтобы ссылка не наследовала поведение усиления, действующее в других ссылках, и, следовательно, не выдавала запросы AJAX. Необходимость «нормального» поведения ссылки объясняется тем, что запрос AJAX не может загрузить файл напрямую, как это делает простой якорный тег.

Загрузка готового архива

Последним шагом станет обработка запроса `GET` к `/contacts/archive/file`. Файл, созданный архиватором, отправляется клиенту. Нам повезло: во Flask существует механизм отправки файла как загруженного ответа: метод `send_file()`.

Как видно из приведенного ниже кода, при вызове `send_file()` передаются три аргумента: путь к архивному файлу, созданному архиватором; имя файла, который должен быть создан браузером; и признак отправки «как вложения». Последний аргумент указывает Flask на необходимость присвоить заголовку ответа HTTP `Content-Disposition` значение `attachment` с заданным файлом; именно этот признак инициирует поведение загрузки файла в браузере.

Листинг 121. Отправка файла клиенту

```
@app.route("/contacts/archive/file", methods=["GET"])
def archive_content():
  manager = Archiver.get()
  return send_file(manager.archive_file(), "archive.json",
as_attachment=True) ❶
```

❶ Отправляет файл клиенту методом Flask send_file().

Великолепно. Мы создали очень элегантный UI архивации. Когда пользователь щелкает на кнопке Download Contacts Archive (Загрузить архив контактов), выводится индикатор прогресса. По достижении 100 % индикатор исчезает, и выводится ссылка на загрузку файла архива. Пользователь может щелкнуть на ссылке, чтобы загрузить архив.

Интерфейс, предлагаемый нашим приложением, намного более удобен для пользователя, чем применяемая на многих сайтах схема «щелкни и жди».

Сглаживание: анимация в htmx

Каким бы симпатичным ни был интерфейс, в нем остается один небольшой недостаток: при обновлении индикатор прогресса «перепрыгивает» от текущей позиции к следующей. Это напоминает полное обновление страницы в приложениях в стиле Web 1.0. Можно ли это исправить? (Разумеется, можно, не зря же мы использовали `div` вместо элемента `progress`!)

Разберем причины этой визуальной проблемы и ее возможные решения. (Если вам не терпится увидеть ответ, можете переходить вперед к «нашему решению».)

Оказывается, в HTML существует встроенная технология для сглаживания перехода элементов между состояниями: API переходов CSS, который уже обсуждался в главе 4. Благодаря переходам CSS можно выполнить плавную анимацию элемента между разными стилями, используя свойство `transition`.

Взглянув на определение CSS класса `.progress-bar`, вы увидите следующее определение перехода: `transition: width .6s ease;`. Это означает, что, когда ширина индикатора прогресса изменяется, скажем, с 20 до 30 %, браузер воспроизводит анимацию за 0,6 секунды с использованием функции `ease` (которая имеет приятный эффект ускорения/замедления).

Почему этот переход не применяется в текущем интерфейсе? Дело в том, что в нашем примере `htmx` *заменяет* индикатор прогресса новым экземпляром при каждом опросе. Это происходит без изменения ширины *существующего* элемента. К сожалению, переходы CSS применяются только при непосредственном изменении свойств существующего элемента, а не при его замене.

Из-за этого приложения на основе «чистого» HTML могут выглядеть дергаными и недостаточно плавными по сравнению с аналогичными приложениями SPA: переходы CSS трудно использовать без JavaScript.

Впрочем, есть и хорошие новости: `htmx` предоставляет возможность использовать переходы CSS даже при замене контента в DOM.

«Стабилизация» в htmx

При обсуждении модели замены `htmx` в главе 4 мы сосредоточились на классах, которые `htmx` добавляет и удаляет, но при этом опустили процесс «стабилизации» (settling). В `htmx` стабилизация состоит из нескольких шагов: когда библиотека `htmx` собирается заменить блок контента, она просматривает новый контент и находит все элементы, для которых определен идентификатор (`id`). Затем она ищет в *существующем* контенте элементы с такими же идентификаторами.

Если такой элемент находится, выполняется довольно затейливая процедура.

- *Новый* контент временно получает атрибуты *старого* контента.
- Происходит вставка нового контента.
- После короткой задержки атрибутам нового контента возвращаются их исходные значения.

К чему эти странные телодвижения?

Если у элемента сохраняется стабильный идентификатор между заменами, можно написать переходы CSS между состояниями. Так как *новый* контент ненадолго получает *прежние* атрибуты, механизм переходов CSS активизируется при восстановлении фактических значений.

Наше решение для плавного изменения индикатора

Итак, мы подошли к предлагаемому решению.

Все, что нужно сделать, — добавить стабильный идентификатор в элемент `progress-bar`.

Листинг 122. Плавное изменение индикатора

```
<div class="progress" >
  <div id="archive-progress" class="progress-bar" role="progressbar"
    aria-valuenow="{{ archiver.progress() * 100}}"
    style="width:{{ archiver.progress() * 100 }}%"></div> ❶
</div>
```

❶ Элемент div индикатора прогресса теперь сохраняет стабильный идентификатор между запросами.

Несмотря на сложную механику, которая работает за кулисами `htmx`, решение сводится к простому добавлению стабильного атрибута `id` к элементу, к которому вы хотите применить анимацию.

Теперь вместо того, чтобы рывком смещаться при каждом обновлении, индикатор прогресса плавно движется по экрану при обновлении, используя переход

CSS, определенный в таблице стилей. Модель замены `htmx` позволяет добиться этого поведения, даже притом что мы заменяем контент новой разметкой HTML.

Итак, мы создали симпатичный индикатор прогресса с плавной анимацией для функциональности архивации контактов. Результат выглядит и ведет себя как инструмент на основе JavaScript, но получен он с использованием простых решений на основе HTML.

И теперь, дорогой читатель, у вас есть повод гордиться собой.

Закрытие интерфейса загрузки

Некоторые пользователи могут передумать и отказаться от загрузки архива. Они никогда не увидят, как движется наш чудесный индикатор, но это их проблема. Мы предоставим таким пользователям кнопку, которая закрывает ссылку на загрузку и возвращает UI к исходному состоянию.

Для этого добавим кнопку, которая выдает запрос `DELETE` к пути `/contacts/archive`, что означает, что текущий архив может быть удален.

Мы добавим кнопку после ссылки на загрузку.

Листинг 123. Скрытие кнопки загрузки

```
  <a hx-boost="false" href="/contacts/archive/file">Archive Ready! Click
here to download. &downarrow;</a>
  <button hx-delete="/contacts/archive">Clear Download</button> ❶
```

❶ Простая кнопка, которая выдает запрос DELETE к /contacts/archive.

Теперь у пользователя есть кнопка, щелчок по которой закрывает ссылку на загрузку архива. Впрочем, ее еще нужно связать со стороной сервера. Как обычно, это довольно просто: мы создаем новый обработчик для действия HTTP `DELETE`, вызываем метод `reset()` для архиватора и заново рендерим шаблон archive_ui.html.

Так как для этой кнопки выбирается та же конфигурация `hx-target` и `hx-swap`, как и для всего остального, она «просто работает».

Код на стороне сервера выглядит так:

Листинг 124. Обработчик отмены загрузки

```
@app.route("/contacts/archive", methods=["DELETE"])
def reset_archive():
  archiver = Archiver.get()
  archiver.reset() ❶
  return render_template("archive_ui.html", archiver=archiver)
```

❶ Вызывает reset() для архиватора.

Похоже на другие обработчики, не так ли?

Да, так! Именно это и было нужно!

Альтернативный вариант UX: автозагрузка

Хотя текущее решение для архивации контактов кажется нам оптимальным, возможны и другие варианты. Сейчас индикатор прогресса показывает ход процесса, а по завершении архивации пользователь получает ссылку для фактической загрузки файла.

Еще один паттерн, который встречается в веб-среде, — «автозагрузка». С этим механизмом файл загружается сразу, не требуя щелчка по ссылке.

Эту функциональность легко добавить в приложение, написав небольшой скрипт. Скриптовый код в гипермедиа-управляемых приложениях более подробно рассматривается в главе 9, но если кратко: скрипты полностью приемлемы в HDA при условии, что они не заменяют базовую гипермедиа-механику приложения.

Для функции автозагрузки мы выбрали скрипт _hyperscript[1]. JavaScript здесь также будет работать и почти так же просто; и снова мы подробно обсудим скриптовые решения в главе 9.

Для реализации функциональности автозагрузки необходимо сделать совсем немного: по окончании рендеринга ссылки автоматически щелкнуть на ней вместо пользователя.

Код _hyperscript читается практически так же, как и предыдущее предложение (и это главная причина, по которой нам так нравится _hyperscript).

Листинг 125. Автозагрузка

```
<a hx-boost="false" href="/contacts/archive/file"
  _="on load click() me"> ❶
  Archive Downloading! Click here if the download does not start.
</a>
```

❶ Фрагмент _hyperscript, обеспечивающий автозагрузку файла.

Здесь принципиально то, что скрипт просто *расширяет* существующую функциональность гипермедиа вместо того, чтобы заменять ее негипермедийным запросом. Это скриптовое расширение, дружественное гипермедиа, о чем подробно мы поговорим ниже.

[1] https://hyperscript.org/

UI динамической архивации готов

В этой главе мы построили динамический UI для архивации контактов, с индикатором прогресса и автозагрузкой, и почти все — за исключением небольшого фрагмента скрипта для автозагрузки — было реализовано на уровне чистого гипермедиа. Для этого потребовалось всего 16 строк кода фронтенда и 16 строк кода бэкенда.

HTML с гипермедиа-ориентированной библиотекой JavaScript (такой, как `htmx`) может быть чрезвычайно мощным и выразительным средством.

Заметки об HTML: каша из Markdown

Каша из Markdown — менее известный «родственник» каши из `<div>`. Она возникает в результате того, что веб-разработчики ограничиваются набором элементов, для которых язык Markdown предоставляет сокращенную запись, даже если эти элементы плохо подходят для конкретной задачи. Серьезный разработчик должен знать весь потенциал используемых инструментов, включая HTML. Возьмем следующий пример оформления ссылок в стиле IEEE.

```
[1] C.H. Gross, A. Stepinski, and D. Akşimşek, ❶
_Hypermedia Systems_, ❷
Bozeman, MT, USA: Big Sky Software.
Available: <https://hypermedia.systems/>
```

❶ Номер ссылки записывается в квадратных скобках.

❷ Символы подчеркивания, в которых заключается название книги, создают элемент .

Здесь `<em>` используется потому, что это единственный элемент Markdown, по умолчанию выводимый курсивом. Это означает, что акцентируется название книги, однако цель в том, чтобы обозначить, что это название работы. В HTML существует элемент `<cite>`, предназначенный именно для этой цели.

Более того, хотя этот нумерованный список идеально подходит для элемента `<ol>`, поддерживаемого Markdown, для номеров ссылок используется простой текст. Почему? Стиль оформления ссылок IEEE требует, чтобы числа заключались в квадратные скобки. Этого можно добиться элементом `<ol>` с CSS, но в Markdown не предусмотрена возможность добавления классов к элементам, а это означает, что квадратные скобки будут применяться ко всем упорядоченным спискам.

Не отказывайтесь от использования встроенного HTML в Markdown. Для больших сайтов также рассмотрите возможность использования расширений Markdown.

```
{.ieee-reference-list} ❶
1. C.H. Gross, A. Stepinski, and D. Akşimşek, ❷
 <cite>Hypermedia Systems</cite>, ❸
 Bozeman, MT, USA: Big Sky Software.
 Available: <https://hypermedia.systems/>
```

❶ Многие диалекты Markdown позволяют добавлять идентификаторы, классы и атрибуты в фигурных скобках.

❷ Теперь можно использовать элемент <ol> и создать скобки в CSS.

❸ <cite> используется для пометки названия цитируемой книги (не всей ссылки!).

Также можно воспользоваться настраиваемым процессором для генерирования подробно детализированной разметки HTML, чтобы не писать ее вручную.

```
{% reference_list %} ❶
[hypers2023]: ❷
C.H. Gross, A. Stepinski, and D. Akşimşek, _Hypermedia Systems_,
Bozeman, MT, USA: Big Sky Software, 2023.
Available: <https://hypermedia.systems/>
{% end %}
```

❶ reference_list — макрос, преобразующий простой текст в детализированную разметку HTML.

❷ Процессор также может автоматически разрешать идентификаторы, чтобы не приходилось вручную упорядочивать ссылки и синхронизировать внутритекстовые ссылки.

ХИТРЫЕ ПРИЕМЫ HTMX

Расширенные возможности htmx

В этой главе мы еще больше углубимся в инструменты `htmx`. Даже с теми из них, что вы уже узнали, можно сделать очень много. Тем не менее при разработке приложений HDA встречаются ситуации, в которых приходится искать дополнительные решения и средства.

В этой главе будут рассмотрены более сложные атрибуты `htmx`, а также продвинутые возможности использованных ранее атрибутов.

Кроме того, будет рассмотрена функциональность, предоставляемая `htmx` помимо простых атрибутов HTML: как `htmx` расширяет стандартные запросы и ответы HTTP, как `htmx` работает с событиями (и генерирует их) и что делать, если на странице нет простого отдельного целевого элемента, который можно обновить.

Наконец, мы рассмотрим некоторые практические вопросы разработки `htmx`: как эффективно заниматься отладкой приложения на основе `htmx`, какие соображения безопасности необходимо учитывать при работе с `htmx` и как настраивать поведение `htmx`.

С инструментами и возможностями, описанными в этой главе, вы сможете создавать довольно сложные пользовательские интерфейсы, используя только `htmx` и, возможно, некоторые подходящие для гипермедиа скрипты на стороне клиента.

Атрибуты htmx

До сих пор мы использовали около 15 атрибутов `htmx`. Самыми важными из них были:

- `hx-get`, `hx-post` и т. д. — для определения запроса AJAX, который должен инициироваться элементом;
- `hx-trigger` — для определения события, инициирующего запрос;

- `hx-swap` — для описания подстановки возвращаемого контента HTML в DOM;
- `hx-target` — для определения того, в какой позиции DOM должен подставляться возвращаемый контент HTML.

У двух из этих атрибутов, `hx-swap` и `hx-trigger`, существует ряд полезных разновидностей для создания более современных гипермедиа-управляемых приложений.

hx-swap

Начнем с атрибута `hx-swap`. Он часто не включается в элементы, выдающие запросы на основе `htmx`, потому что поведение по умолчанию — `innerHTML`, заменяющее внутреннюю разметку HTML элемента, — подходит для большинства практических сценариев.

Мы уже познакомились с ситуациями, в которых требовалось переопределить поведение по умолчанию и использовать, например, `outerHTML`. В главе 2 также были представлены другие варианты замены — `beforebegin`, `afterend` и т. д.

В главе 5 мы рассмотрели модификатор задержки `swap` для `hx-swap`, при помощи которого можно воспроизвести эффект постепенного скрытия контента перед его удалением из DOM.

Кроме того, `hx-swap` предлагает дополнительные возможности управления со следующими модификаторами:

- `settle` — как и `swap`, позволяет применить установленную задержку между моментом замены контента в DOM и «стабилизацией» его атрибутов (то есть обновления их прежних значений новыми значениями). Это дает возможность точного управления переходами CSS;
- `show` — позволяет задать элемент, который должен быть показан (с возможной прокруткой в область просмотра браузера, если потребуется) при завершении запроса;
- `scroll` — позволяет задать прокручиваемый элемент (то есть элемент с полосами прокрутки), который должен быть прокручен к верхней или нижней границе при завершении запроса;
- `focus-scroll` — позволяет указать, что при завершении запроса `htmx` следует выполнить прокрутку к элементу, обладающему фокусом. По умолчанию этот модификатор установлен в false.

Таким образом, например, если имеется кнопка, выдающая запрос `GET`, и необходимо, чтобы при завершении запроса выполнялась прокрутка к верхней границе элемента `body`, напишите следующую разметку HTML.

Листинг 126. Прокрутка к верху страницы

```
<button hx-get="/contacts" hx-target="#content-div"
    hx-swap="innerHTML show:body:top"> ❶
 Get Contacts
</button>
```

❶ Сообщает htmx показать верхнюю границу body после замены.

За более подробной информацией и другими примерами обращайтесь к электронной документации `hx-swap`[1].

hx-trigger

Как и `hx-swap`, атрибут `hx-trigger` часто опускается при работе с `htmx`, потому что обычно востребовано его поведение по умолчанию. Напомним, что инициирующие события по умолчанию определяются типом элемента:

- запросы элементов `input`, `textarea` и `select` инициируются событием `change`;
- запросы элементов `form` инициируются событием `submit`;
- запросы всех остальных элементов инициируются событием `click`.

Впрочем, иногда требуется более точно управлять инициирующими событиями. Классический пример — активный поиск, реализованный в Contact.app.

Листинг 127. Поле ввода с активным поиском

```
  <input id="search" type="search" name="q" value="{{ request.args.get('q')
or '' }}"
     hx-get="/contacts"
     hx-trigger="search, keyup delay:200ms changed"/> ❶
```

❶ Расширенная спецификация триггера.

В этом примере используются два модификатора, доступных для атрибута `hx-trigger`:

- `delay` — задает задержку перед выдачей запроса. Если событие произойдет повторно в течение заданного интервала, то первое событие отбрасывается и таймер обнуляется. Это позволяет устранять «дребезг» события.
- `changed` — указывает, что запрос должен выдаваться только при изменении свойства `value` заданного элемента.

`Hx-trigger` — поддерживает ряд дополнительных модификаторов И это понятно, потому что система событий довольно сложна и необходимо иметь возмож-

[1] https://htmx.org/attributes/hx-swap/

ность задействовать весь ее потенциал. События будут более подробно рассмотрены ниже.

Другие модификаторы, доступные для `hx-trigger`:

- `once` — заданное событие инициирует запрос только один раз.
- `throttle` — позволяет регулировать события, выдавая их не чаще одного раза в заданный промежуток времени. В отличие от `delay`, первое событие инициируется немедленно, но все последующие события будут инициироваться только по завершении времени регулировки.
- `from` — селектор CSS, который позволяет выбрать другой элемент для прослушивания событий. Пример использования будет приведен позже в этой главе.
- `target` — селектор CSS, позволяющий фильтровать только те события, которые происходят в границах заданного элемента. В DOM события «всплывают» к своим родительским элементам, так что событие `click` для кнопки также будет инициировать событие `click` для родительского элемента `div`, и так на всем пути к элементу `body`. Иногда требуется задать событие непосредственно для конкретного элемента, и модификатор `target` позволяет это сделать.
- `consume` — если модификатор имеет значение `true`, то инициирующее событие будет отменено и не будет распространяться в родительские элементы.
- `queue` — определяет, как должны формироваться очереди событий в `htmx`. По умолчанию при получении события-триггера `htmx` выдает запрос и запускает очередь событий. Если запрос еще не обработан на момент получения следующего события, то событие помещается в очередь, а при завершении запроса выдается новый запрос. По умолчанию в очереди хранится только последнее полученное событие, но такое поведение можно изменить при помощи этого модификатора: например, можно присвоить значение `none` и игнорировать все события-триггеры, происходящие во время обработки запроса.

Фильтры триггеров

Атрибут `hx-trigger` также позволяет задать *фильтр* для событий. Для этого после имени события в квадратных скобках указывается выражение JavaScript.

Представим сложный сценарий, в котором получение контактов должно быть разрешено только при определенных условиях. Имеется функция JavaScript `contactRetrievalEnabled()`, которая возвращает логический признак: `true`, если получение контактов разрешено, и `false` в остальных случаях. Как использовать эту функцию для ограничения доступа к кнопке, выдающей запрос к `/contacts`?

Чтобы решить эту задачу с использованием фильтра событий в `htmx`, напишите следующую разметку HTML.

Листинг 128. Поле ввода с активным поиском

```
<script>
 function contactRetrievalEnabled() {
   // Код, проверяющий, разрешено ли получение контактов
   ...
 }
</script>
<button hx-get="/contacts" hx-trigger="click[contactRetrievalEnabled()]"> ❶
 Get Contacts
</button>
```

❶ Запрос выдается по событию click только в том случае, если contactRetrievalEnabled() возвращает true.

Кнопка не выдает запрос, если `contactRetrievalEnabled()` возвращает `false`, что позволяет динамически управлять возможностью выдачи запросов. Многие типичные сценарии требуют использования триггера события, тогда как запрос должен выдаваться только в определенных обстоятельствах:

- если определенный элемент обладает фокусом;
- если заданная форма содержит проверенные данные;
- если группа полей ввода содержит конкретные значения.

Фильтры событий позволяют применить любую нужную логику для фильтрации запросов `htmx`.

Синтетические события

Кроме перечисленных модификаторов, `hx-trigger` предоставляет несколько «синтетических» событий, то есть событий, не являющихся частью обычного DOM API. События `load` и `revealed` уже встречались в примерах отложенной загрузки и бесконечной прокрутки, но `htmx` также предоставляет событие `intersect`, которое срабатывает при пересечении элемента с его родительским элементом.

Это синтетическое событие использует современный API наблюдателей пересечения, о котором можно больше узнать из MDN[1].

Пересечения предоставляют возможность более точно контролировать, когда именно должен инициироваться запрос. Например, можно установить порог

[1] https://developer.mozilla.org/en-US/docs/Web/API/Intersection_Observer_API

и указать, что запрос должен выдаваться только в том случае, если элемент виден на 50 %.

Безусловно, `hx-trigger` — самый сложный из атрибутов `htmx`. За более подробной информацией о нем и другими примерами обращайтесь к электронной документации[1].

Другие атрибуты

`Htmx` предоставляет множество других, реже используемых атрибутов для настройки поведения гипермедиа-управляемых приложений.

Ниже перечислены самые полезные из этих атрибутов:

- `hx-push-url` — «проталкивает» URL запроса (или другое значение) в адресную строку.
- `hx-preserve` — сохраняет часть DOM между запросами; исходный контент будет сохранен независимо от того, что будет возвращено.
- `hx-sync` — синхронизирует запросы между двумя и более элементами.
- `hx-disable` — отключает поведение `htmx` для этого элемента и всех его дочерних элементов. Мы вернемся к этой теме, когда будем обсуждать проблему безопасности.

Рассмотрим подробнее атрибут `hx-sync`, который позволяет синхронизировать запросы AJAX между двумя и более элементами. Возьмем простой случай с двумя кнопками, целевым для которых является один и тот же элемент на экране.

Листинг 129. Две конкурирующие кнопки

```
<button hx-get="/contacts" hx-target="body">
 Get Contacts
</button>
<button hx-get="/settings" hx-target="body">
 Get Settings
</button>
```

Все работает нормально, но что, если пользователь щелкнет на кнопке Get Contacts (Контакты), а ответ на запрос будет получен не сразу? И за это время он успеет щелкнуть по кнопке Get Settings (Настройки)? В таком случае будут существовать два необработанных запроса одновременно.

Предположим, запрос `/settings` завершается первым и выводит информацию о настройках. Пользователь начинает вносить изменения, но тут его ждет сюр-

[1] https://htmx.org/attributes/hx-trigger/

приз — запрос к `/contacts` завершается и заменяет все тело документа информацией о контактах!

Чтобы решить эту проблему, можно воспользоваться `hx-indicator` и уведомить пользователя о том, что выполняется определенный процесс, снижая вероятность щелчка второй кнопки. Но если необходимо действительно предусмотреть, чтобы в любой момент времени между этими кнопками существовал только один запрос, следует воспользоваться атрибутом `hx-sync`. Заключим обе кнопки в тег `div` и исключим лишнюю спецификацию `hx-target`, переместив атрибут вверх в `div`. Затем можно использовать `hx-sync` с `div`, чтобы координировать запросы между двумя кнопками.

Обновленный код выглядит так:

```
<div hx-target="body" ❶
  hx-sync="this"> ❷
  <button hx-get="/contacts"> ❶
   Get Contacts
  </button>
  <button hx-get="/settings"> ❶
   Get Settings
  </button>
</div>
```

❶ Повторяющиеся атрибуты hx-target поднимаются в родительский элемент div.
❷ Синхронизация по родительскому div.

Размещая в `div` атрибут `hx-sync` со значением `this`, мы говорим: «Синхронизировать все запросы `htmx`, инициируемые в этом элементе `div`, по отношению друг к другу». Это означает, что, если одна кнопка уже выдала незавершенный запрос, другие кнопки внутри `div` не смогут выдавать запросы до его завершения.

Атрибут `hx-sync` также поддерживает другие стратегии, которые позволяют, например, заменить существующий запрос «на лету» или ставить запросы в очередь с конкретной стратегией организации очереди. Полную документацию вместе с примерами можно найти на странице `hx-sync`[1] на сайте htmx.org.

Как видите, `htmx` предлагает значительную функциональность, управляемую атрибутами, для более мощных гипермедиа-управляемых приложений. Полную справку по всем атрибутам `htmx` можно найти на сайте `htmx`[2].

[1] https://htmx.org/attributes/hx-sync/

[2] https://htmx.org/reference/#attributes

События

До сих пор мы работали с событиями JavaScript в `htmx` в основном через атрибут `hx-trigger`. Этот атрибут предоставляет эффективный механизм управления приложением с использованием декларативного синтаксиса, хорошо сочетающегося с HTML.

Тем не менее на этом возможности событий не исчерпаны. События играют важнейшую роль в расширении HTM как среды гипермедиа, а также, как вы вскоре убедитесь, в дружественных ей скриптах.

События — это «клей», связывающий воедино DOM, HTML, `htmx` и скрипты. Можно даже рассматривать DOM как сложную «шину событий» для приложений.

Очень важно: чтобы строить продвинутые гипермедиа-управляемые приложения, необходимо хорошо разобраться в событиях. Поверьте, вы не пожалеете о потраченном времени[1].

События, генерируемые htmx

Кроме простой *обработки* событий, `htmx` *генерирует* много полезных событий. Эти события могут использоваться для добавления новой функциональности в приложения — либо через саму библиотеку `htmx`, либо через скрипты.

Некоторые события, чаще всего генерируемые в `htmx`:

- `htmx:load` — инициируется при загрузке нового контента в DOM библиотекой `htmx`;
- `htmx:configRequest` — инициируется перед выдачей запроса, позволяя запрограммировать запрос или полностью отменить его;
- `htmx:afterRequest` — инициируется после ответа на запрос;
- `htmx:abort` — нестандартное событие, которое может быть отправлено элементу на основе `htmx` для отмены открытого запроса.

Использование события htmx:configRequest

Рассмотрим пример использования событий, генерируемых `htmx`. Мы воспользуемся событием `htmx:configRequest` для настройки запроса HTTP.

Представьте следующую ситуацию: команда, пишущая код на стороне сервера, решила, что для повышения безопасности в каждый запрос должен включаться маркер, сгенерированный сервером. Маркер будет храниться в локальном хранилище (`localStorage`) браузера, в слоте `special-token`.

---

[1] https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/Events

Маркер будет назначаться кодом JavaScript (пока не думайте о подробностях) при первом входе пользователя в приложение.

Листинг 130. Получение маркера в JavaScript

```
  let response = await fetch("/token"); ❶
  localStorage['special-token'] = await response.text();
```

❶ Получает значение маркера, а затем сохраняет его в localStorage.

Серверная команда требует, чтобы вы включали этот специальный маркер при каждом запросе, выполняемом `htmx`, в заголовке `X-SPECIAL-TOKEN`. Как этого добиться? Один из способов основан на перехвате события `htmx:configRequest` и обновлении объекта `detail.headers` маркером из `localStorage`.

В «базовом» JS это будет выглядеть примерно так (код включается в тег `<script>` в теге `<head>` документа HTML):

Листинг 131. Добавление заголовка X-SPECIAL-TOKEN

```
document.body.addEventListener("htmx:configRequest", function(configEvent){
  configEvent.detail.headers['X-SPECIAL-TOKEN'] = localStorage['special-
token']; ❶
})
```

❶ Получает значение из локального хранилища и записывает его в заголовок.

Как видите, мы добавляем новое значение в свойство `headers` свойства `detail` события. После выполнения обработчика события свойство `headers` читается `htmx` и используется для построения заголовков генерируемого запроса AJAX.

Свойство `detail` события `htmx:configRequest` содержит полезные атрибуты, которые можно обновлять для изменения «структуры» запроса, в том числе:

- `detail.parameters` — позволяет добавлять или удалять параметры запроса;
- `detail.target` — позволяет обновить цель запроса;
- `detail.verb` — позволяет обновить «команду» HTTP запроса (то есть `GET`).

Например, если серверная команда решит, что маркер должен быть добавлен в виде параметра (вместо заголовка запроса), код можно привести к следующему виду.

Листинг 132. Добавление параметра token

```
document.body.addEventListener("htmx:configRequest", function(configEvent){
  configEvent.detail.parameters['token'] = localStorage['special-token'];
❶
})
```

❶ Получает значение из локального хранилища и присваивает его параметру.

Как видите, это решение обеспечивает значительную гибкость при обновлении запроса AJAX, выдаваемого htmx.

Полная документация по событию htmx:configRequest (и другим событиям, которые могут вас заинтересовать) находится на сайте htmx[1].

Отмена запроса с использованием htmx:abort

В htmx можно прослушивать множество полезных событий и реагировать на них при помощи hx-trigger. А что еще можно делать с событиями?

Сама библиотека htmx прослушивает одно специальное событие htmx:abort. Когда htmx получает это событие для незавершенного запроса, этот запрос отменяется.

Рассмотрим сценарий, в котором имеется потенциально долго выполняющийся запрос к /contacts и необходимо предоставить пользователю возможность отменить этот запрос. Для этого нужны кнопка, выдающая запрос (конечно, под управлением htmx), и другая кнопка, которая отправляет событие htmx:abort первой кнопке.

Код может выглядеть примерно так:

Листинг 133. Кнопка с возможностью отмены

```
<button id="contacts-btn" hx-get="/contacts" hx-target="body"> ❶
 Get Contacts
</button>
<button onclick="document.getElementById('contacts-btn').dispatchEvent(new
Event('htmx:abort'))"> ❷
 Cancel
</button>
```

❶ Обычный запрос GET на основе htmx к /contacts.

❷ Код JavaScript для поиска кнопки и отправки ей события htxm:abort.

Если пользователь щелкает на кнопке Get Contacts и запрос занимает некоторое время, пользователь может щелкнуть на кнопке Cancel и отменить запрос. Конечно, в более сложном пользовательском интерфейсе кнопка Cancel должна блокироваться при отсутствии незавершенного запроса HTTP, но реализация этой функциональности на «чистом» HTML будет слишком хлопотной.

К счастью, в _hyperscript реализация достаточно проста. Результат будет выглядеть примерно так:

[1] https://htmx.org/events/#htmx:configRequest

Листинг 134. Кнопка с возможностью отмены на основе _hyperscript

```
<button id="contacts-btn" hx-get="/contacts" hx-target="body">
 Get Contacts
</button>
<button _="on click send htmx:abort to #contacts-btn
     on htmx:beforeRequest from #contacts-btn remove @disabled from me
     on htmx:afterRequest from #contacts-btn add @disabled to me">
 Cancel
</button>
```

Теперь кнопка Cancel блокируется только при наличии незавершенного запроса от кнопки `contacts-btn`. И чтобы это решение работало, мы используем события, генерируемые и обрабатываемые средствами `htmx`, а также синтаксис _hyperscript, хорошо сочетающийся с событиями.

События, генерируемые сервером

В следующем разделе мы продолжим разговор о том, как `htmx` расширяет обычные запросы и ответы HTTP, но так как этот механизм основан на событиях, рассмотрим один заголовок ответа HTTP, поддерживаемый `htmx`: `HX-Trigger`. Ранее вы узнали, как запросы и ответы HTTP поддерживают *заголовки* — пары «имя-значение», содержащие метаданные о запросе или ответе. В частности, мы рассмотрели заголовок запроса `HX-Trigger`, включающий идентификатор элемента, инициирующего заданный запрос.

Кроме *заголововвка запроса*, `htmx` поддерживает *заголовок ответа* с именем `HX-Trigger`. Заголовок ответа позволяет *инициировать событие* для элемента, отправившего запрос AJAX. Это мощный механизм координации элементов в DOM без сильной связанности.

Чтобы понять, как он работает, рассмотрим следующий сценарий: имеется кнопка, которая получает новые контакты с удаленной системы на сервере. Подробности реализации на стороне сервера нас сейчас не интересуют, но мы знаем, что при выдаче запроса `POST` к пути `/sync` будет инициирована синхронизация с системой.

Синхронизация может привести (а может и не привести) к созданию новых контактов. *При создании* новых контактов необходимо обновить таблицу контактов. Если же контакты не создаются, то обновлять таблицу не нужно.

Чтобы реализовать эту функциональность, можно по условию добавить заголовок ответа `HX-Trigger` со значением `contacts-updated`.

Листинг 135. Условное инициирование события contacts-updated

```
@app.route('/sync', methods=["POST"])
def sync_with_server():
```

```
contacts_updated = RemoteServer.sync() ❶
resp = make_response(render_template('sync.html'))
if contacts_updated ❷
 resp.headers['HX-Trigger'] = 'contacts-updated'
return resp
```

❶ Вызов к удаленной системе, который синхронизирует базу данных контактов.

❷ Если какие-либо контакты были обновлены, клиент инициирует событие contacts-updated по условию.

Это значение инициирует событие `contacts-updated` для кнопки, которая выдает запрос AJAX к `/sync`. Затем можно воспользоваться модификатором `from:` атрибута `hx-trigger` для прослушивания этого события. С таким паттерном можно фактически инициировать запросы `htmx` со стороны сервера.

Код на стороне клиента может выглядеть так:

Листинг 136. Таблица Contacts

```
<button hx-post="/integrations/1"> ❶
 Pull Contacts From Integration
</button>

  ...

<table hx-get="/contacts/table" hx-trigger="contacts-updated from:body"> ❷
 ...
</table>
```

❶ Ответ на этот запрос может инициировать событие contacts-updated по условию.

❷ Таблица прослушивает событие и обновляется при его возникновении.

Таблица прослушивает событие `contacts-updated`, причем делает это на уровне элемента `body`. Она прослушивает элемент `body`, так как событие всплывает вверх от кнопки, и это позволяет избежать сильной связанности кнопки с таблицей: кнопку и таблицу можно перемещать как угодно, но благодаря событиям нужное поведение продолжит нормально работать. Кроме того, может потребоваться, чтобы событие `contacts-updated` инициировалось *другими* элементами или запросами, поэтому эта схема обеспечивает обобщенный механизм обновления таблицы контактов в приложении.

Запросы и ответы HTTP

Вы уже видели одну расширенную возможность ответов HTTP, поддерживаемую `htmx`, — заголовок ответа `HX-Trigger`; но `htmx` поддерживает и другие заголовки запросов и ответов. В главе 4 рассматривались заголовки, присутствующие в за-

просах HTTP. Вот некоторые важные заголовки, используемые для изменения поведения `htmx` с ответами HTTP:

- `HX-Location` — инициирует перенаправление на стороне клиента;
- `HX-Push-Url` — заносит новый URL в адресную строку браузера;
- `HX-Refresh` — обновляет текущую страницу;
- `HX-Retarget` — позволяет задать новую цель для замены контента ответа на стороне клиента.

За информацией обо всех заголовках запросов и ответов обращайтесь к документации `htmx`[1].

Коды ответов HTTP

Пожалуй, в отношении информации, передаваемой клиенту, *коды ответов HTTP* даже важнее заголовков ответов. Коды ответов HTTP рассматривались в главе 3. В целом `htmx` обрабатывает коды ответов ожидаемым образом: для всех кодов ответов уровня 200 выполняется замена контента, а для остальных ничего не происходит. Тем не менее существуют два «специальных» кода ответов уровня 200:

- 204 No Content (Нет содержимого) — при получении этого кода ответа `htmx` не подставляет контент в DOM (даже если у ответа есть тело);
- 286 — при получении этого кода ответа на запрос, для которого выполняется опрос (polling), `htmx` прерывает опрос.

Поведение `htmx` в отношении кодов ответов можно переопределить — как вы уже догадались, реакцией на событие! Событие `htmx:beforeSwap` позволяет изменить поведение `htmx` в отношении разных кодов состояния.

Предположим, что вместо того, чтобы ничего не делать при получении кода 404, вы хотите оповестить пользователя о возникшей ошибке при помощи вызова метода JavaScript `showNotFoundError()`. Для этого добавим следующий фрагмент кода в событие `htmx:beforeSwap`.

Листинг 137. Вывод диалогового окна 404

```
document.body.addEventListener('htmx:beforeSwap', function(evt) { ❶
  if(evt.detail.xhr.status === 404){ ❷
    showNotFoundError();
  }
});
```

❶ Подключение к событию htmx:beforeSwap.

❷ Если код ответа равен 404, вывести диалоговое окно.

[1] https://htmx.org/reference/#headers

Для определения того, должен ли ответ подставляться в DOM и какой элемент назначать целевым для ответа, также можно воспользоваться событием `htmx:beforeSwap`. Оно обеспечивает некоторую свободу выбора способа использования кодов ответов HTTP в приложении. Полная документация для события `htmx:beforeSwap` доступна на сайте htmx.org[1].

Обновление остального контента

Выше мы увидели, как использовать событие, инициируемое сервером через заголовок ответа `HX-Trigger`, чтобы обновить блок DOM, который основан на ответе другого блока DOM. Этот прием помогает решить общую проблему гипермедиа-управляемых приложений: как обновить остальной контент? Ведь в обычных запросах HTTP существует только одна «цель» — весь экран; аналогичным образом в запросах на основе `htmx` есть только одна цель: явная или неявная цель элемента.

Рассмотрим несколько вариантов того, как обновить другой контент в `htmx`.

Расширение выделения

Первый и самый простой вариант — «расширение цели». Иначе говоря, вместо того чтобы заменять небольшую часть экрана, расширяйте цель запроса на основе `htmx`, пока она не будет включать все элементы, которые должны обновляться на экране. У такого решения есть два огромных преимущества — простота и надежность. Недостаток заключается в том, что оно не предоставляет желаемого взаимодействия с пользователем и не всегда хорошо сочетается с некоторыми макетами шаблонов на стороне сервера. Тем не менее мы всегда рекомендуем для начала хотя бы рассмотреть возможность применения этого подхода.

Внеполосная замена

Второй, чуть более сложный вариант основан на использовании поддержки «внеполосного» (Out Of Band) контента в `htmx`. При получении ответа `htmx` проверяет его и ищет контент верхнего уровня, включающий атрибут `hx-swap-oob`. Этот контент исключается из ответа и не подставляется в DOM обычным образом. Вместо этого он заменяет контент с совпадающим идентификатором.

Рассмотрим конкретный пример. Вспомните ситуацию, описанную выше, когда таблица контактов должна была обновляться при создании новых контактов.

---

[1] https://htmx.org/events/#htmx:beforeSwap

Тогда проблема решалась с использованием событий и событием, инициируемым сервером через заголовок ответа `HX-Trigger`.

На этот раз мы воспользуемся атрибутом `hx-swap-oob` в ответе на запрос `POST` к `/integrations/1`. Новый контент таблицы контактов присоединяется к ответу.

Листинг 138. Обновленная таблица контактов

```
<button hx-post="/integrations/1"> ❶
 Pull Contacts From Integration
</button>

  ...

 <table id="contacts-table"> ❷
  ...
 </table>
```

❶ Кнопка все еще выдает запрос POST к /integrations/1.

❷ Таблица уже не прослушивает событие, но теперь ей назначается идентификатор.

Ответ на запрос `POST` к `/integrations/1` будет включать контент, который должен подставляться в кнопку с использованием обычного механизма `htmx`. Однако он включает новую, обновленную версию таблицы контактов, которая помечается атрибутом `hx-swap-oob="true"`. Контент будет удален из ответа, чтобы он не был вставлен в кнопку. Вместо этого он подставляется в DOM на место существующей таблицы, так как их идентификаторы совпадают.

Листинг 139. Ответ с внеполосным контентом

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
...

Pull Contacts From Integration ❶

<table id="contacts-table" hx-swap-oob="true"> ❷
 ...
</table>
```

❶ Этот контент будет помещен в кнопке.

❷ Этот контент будет удален из ответа и заменен в соответствии с идентификатором.

Используя этот прием, можно обновлять контент страницы там, где потребуется. Атрибут `hx-swap-oob` предоставляет другие возможности, но все они документированы[1].

[1] https://htmx.org/attributes/hx-swap-oob/

В зависимости от того, насколько точно работает технология шаблонов на стороне сервера и какой уровень интерактивности необходим приложению, внеполосная замена может стать мощным механизмом обновления контента.

События

Наконец, самый сложный механизм обновления контента был описан в разделе, посвященном событиям: обновление элементов с помощью событий, инициируемых сервером. Этот механизм может быть оптимальным, но он требует глубокого концептуального знания HTML и событий, а также соблюдения событийного подхода. Хотя нам нравится такой стиль разработки, он не для всех. Обычно мы рекомендуем этот паттерн только в том случае, если философия `htmx` событийного гипермедиа действительно пришлась вам по душе.

Но если она *пришлась* вам по душе — выбирайте ее. Мы создавали очень сложные и гибкие пользовательские интерфейсы, используя этот механизм, и он нам очень нравится.

Прагматичный подход

Все решения проблемы «обновления остального контента» работают, и часто работают хорошо. Однако может наступить момент, когда будет проще выбрать другой подход, например реактивный. Как бы нам ни нравились решения гипермедиа, реальность такова, что некоторые паттерны UX просто невозможно с легкостью реализовать в этой среде. Классическим примером таких паттернов, уже упоминавшимся выше, является электронная онлайн-таблица: ее пользовательский интерфейс слишком сложен и содержит слишком много взаимозависимостей, чтобы его можно было качественно реализовать через обмен контентом гипермедиа с сервером.

В таких случаях (и каждый раз, когда вы чувствуете, что решение на основе `htmx` оказывается более сложным, чем другой подход) мы рекомендуем рассмотреть другие технологии. Будьте прагматичны и выбирайте подходящий инструмент для работы. Вы всегда можете использовать `htmx` для тех частей приложения, которые менее сложны и не требуют всех возможностей реактивного фреймворка, и сэкономить бюджет сложности для других частей.

Мы рекомендуем изучить больше разных веб-технологий, обращая внимание на сильные и слабые стороны каждой. Так у вас сформируется обширный инструментарий, к которому вы сможете обратиться при работе над очередной задачей. По нашему опыту, с `htmx` гипермедиа станет тем инструментом, к которому вы будете обращаться чаще всего.

Отладка

Нам не стыдно признаться: мы большие поклонники событий. Эта технология лежит в основе практически каждого интересного пользовательского интерфейса. После того как вы получите доступ к обобщенному использованию событий в HTML, они станут особенно полезны в DOM. События позволяют строить слабо связанные программные системы, при этом часто сохраняя локальность поведения, которую мы так ценим.

Тем не менее события неидеальны. Одной из областей, в которых события создают особенно много проблем, является *отладка*: часто требуется узнать, почему событие *не* инициируется. Но как установить точку останова для чего-то, что *не происходит*? Никак (по крайней мере, пока).

Существуют два приема, которые могут помочь с отладкой. Первый предоставляется `htmx`, а другой — Chrome, браузером от Google.

Регистрация событий htmx

Первый прием, предоставляемый самой библиотекой `htmx`, заключается в вызове метода `htmx.logAll()`. При этом `htmx` регистрирует в журнале все внутренние события, происходящие при выполнении бизнес-логики, загрузке контента, реакции на события и т. д.

Это может привести к информационной перегрузке, но грамотная фильтрация поможет справиться с проблемой. Вот как выглядит журнал (вернее, его небольшая часть), сохраняемый по щелчку на ссылке «docs» на сайте https://htmx.org при включенном режиме `logAll()`.

Листинг 140. Журнал htmx

```
htmx:configRequest
<a href="/docs/">
Object { parameters: {}, unfilteredParameters: {}, headers: {…}, target:
body, verb: "get", errors: [], withCredentials: false, timeout: 0, path:
"/docs/", triggeringEvent: a
, … }
htmx.js:439:29
htmx:beforeRequest
<a href="/docs/">
Object { xhr: XMLHttpRequest, target: body, requestConfig: {…}, etc: {},
pathInfo: {…}, elt: a
 }
htmx.js:439:29
htmx:beforeSend
<a class="htmx-request" href="/docs/">
```

```
Object { xhr: XMLHttpRequest, target: body, requestConfig: {…}, etc: {},
pathInfo: {…}, elt: a.htmx-request
 }
htmx.js:439:29
htmx:xhr:loadstart
<a class="htmx-request" href="/docs/">
Object { lengthComputable: false, loaded: 0, total: 0, elt: a.htmx-request
 }
htmx.js:439:29
htmx:xhr:progress
<a class="htmx-request" href="/docs/">
Object { lengthComputable: true, loaded: 4096, total: 19915, elt: a.htmx-
request
 }
htmx.js:439:29
htmx:xhr:progress
<a class="htmx-request" href="/docs/">
Object { lengthComputable: true, loaded: 19915, total: 19915, elt: a.htmx-
request
 }
htmx.js:439:29
htmx:beforeOnLoad
<a class="htmx-request" href="/docs/">
Object { xhr: XMLHttpRequest, target: body, requestConfig: {…}, etc: {},
pathInfo: {…}, elt: a.htmx-request
 }
htmx.js:439:29
htmx:beforeSwap
<body hx-ext="class-tools, preload">
```

Не самое легкое чтиво, правда?

Но если сделать глубокий вдох и присмотреться, вы увидите, что не все так плохо: перед вами серия событий `htmx`, и некоторые из них уже вам знакомы (как `htmx:configRequest!`), вместе с элементами, которые их инициировали.

Немного привыкнув к чтению и фильтрации информации в журналах, вы научитесь ориентироваться в потоке событий. Это поможет вам в отладке проблем, связанных с `htmx`.

Мониторинг событий в Chrome

Описанный метод полезен, если проблема где-то *внутри* `htmx`, но что, если `htmx` вообще не инициируется? Такое иногда случается, например, когда вы неправильно ввели имя события.

В таких случаях приходится пользоваться инструментами, доступными в самом браузере. К счастью, браузер Google Chrome предоставляет очень полезную

функцию `monitorEvents()`, которая позволяет наблюдать за *всеми* событиями, инициируемыми для элемента.

Данная возможность доступна *только* в консоли, так что вы не сможете использовать ее в коде страницы. Но если вы работаете с `htmx` в Chrome и вас интересует, почему событие не инициируется для элемента, откройте консоль разработчика и введите следующую команду:

Листинг 141. Мониторинг htmx

```
monitorEvents(document.getElementById("some-element"));
```

Команда выводит *все* события, инициируемые для элемента с идентификатором `some-element`, на консоль. Полученная информация поможет понять, на какие события необходимо реагировать в `htmx`, или разобраться, почему ожидаемое событие не происходит.

Использование этих двух методов поможет диагностировать (хочется надеяться, нечасто) проблемы, связанные с событиями, при разработке с использованием `htmx`.

Соображения безопасности

В общем случае решения на основе `htmx` и гипермедиа оказываются более безопасными, чем подходы к построению веб-приложений, основанные на использовании JavaScript. Дело в том, что благодаря перемещению существенной доли обработки в бэкенд в решениях гипермедиа конечные пользователи обычно не имеют доступа к значительной части системы, чтобы проводить разного рода манипуляции и махинации.

Однако даже в случае использования гипермедиа встречаются ситуации, требующие от разработчика осторожности. Особого внимания заслуживает показ контента, сгенерированного пользователем, другим пользователям: хитрый злоумышленник может вставить в контент код `htmx` и обманом заставить других пользователей щелкнуть на нем, чтобы запустить действия, которые пользователи выполнять не собирались.

В общем случае весь контент, генерируемый пользователем, должен экранироваться на стороне сервера, а большинство фреймворков рендеринга на стороне сервера предоставляют функциональность для обработки таких ситуаций. Однако всегда существует риск упустить какую-нибудь мелочь.

Чтобы упростить жизнь разработчикам, `htmx` предоставляет атрибут `hx-disable`. Если установить этот атрибут для элемента, то все атрибуты `htmx` внутри этого элемента будут игнорироваться.

Политики безопасности контента и htmx

Политика безопасности контента, или CSP (Content Security Policy), — браузерная технология, позволяющая обнаруживать и предотвращать некоторые виды атак, основанные на внедрении контента. Полное обсуждение CSP выходит за рамки этой книги, но мы рекомендуем ознакомиться со статьей в Mozilla Developer Network[1] для получения дополнительной информации.

Типичный пример функциональности, блокируемой CSP, — функция `eval()` в JavaScript, позволяющая выполнить произвольный код JavaScript, содержащийся в строке. Известно, что данная возможность создает риск для безопасности, и многие команды разработчиков решили, что доступность `eval()` в веб-приложениях того не стоит.

В `htmx` функция `eval()` почти не используется, поэтому CSP с таким ограничением будет нормально работать. От `eval()` зависят только фильтры событий, о которых было рассказано выше. Если вы решите заблокировать `eval()` в своем веб-приложении, то не сможете пользоваться синтаксисом фильтрации событий.

Конфигурация

В `htmx` доступны многочисленные параметры конфигурации. Несколько примеров того, что можно настраивать в приложениях:

- стиль замены по умолчанию;
- задержку замены по умолчанию;
- тайм-аут запросов AJAX по умолчанию.

Полный список параметров конфигурации можно найти в соответствующем разделе основной документации `htmx`[2].

`Htmx` обычно настраивается в теге `meta`, находящемся в заголовке страницы. Тегу `meta` должно быть присвоено имя `htmx-config`, а атрибут `content` должен содержать переопределения конфигурации в формате JSON. Пример:

Листинг 142. Конфигурация htmx в теге meta

```
<meta name="htmx-config" content='{"defaultSwapStyle":"outerHTML"}'>
```

В данном случае мы переопределяем стиль замены по умолчанию с обычного `innerHTML` на `outerHTML`. Это может быть полезно, если вы обнаружите, что в ва-

[1] https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP

[2] https://htmx.org/docs/#config

ших приложениях `outerHTML` используется чаще `innerHTML` и лучше избежать необходимости явно задавать это значение.

Заметки об HTML: семантический HTML

Тенденция «использовать семантический HTML» вместо того, чтобы «читать спецификации», привела к тому, что многие стремятся угадать смысл тегов («По-моему, очень семантично!») вместо того, чтобы заглянуть в спецификацию.

> “ Я считаю, что когда вам предлагают писать содержательный HTML, дело вовсе не в смысле текста для человека. Дело в использовании тегов для цели, описанной в спецификации, — удовлетворения потребностей таких программных систем, как браузеры, технологии доступности и поисковые системы.
>
> https://t-ravis.com/post/doc/semantic_the_8_letter_s-word/

Мы рекомендуем обсуждать и писать *соответствующий* HTML. Используйте элементы так, как описано в спецификации HTML, и пусть программные системы извлекают из них смысл на свое усмотрение.

СКРИПТЫ НА СТОРОНЕ КЛИЕНТА

> “ REST позволяет расширять клиентскую функциональность за счет загрузки и выполнения кода в форме апплетов или скриптов. Это упрощает структуру клиента за счет сокращения объема функциональности, которая должна быть реализована заранее.
>
> *Рой Филдинг, «Architectural Styles and the Design of Network-based Software Architectures»*

До сих пор мы (почти) не писали код JavaScript (или _hyperscript) в Contact.app — главным образом потому, что он был не нужен для реализуемой функциональности. В этой главе мы рассмотрим скрипты, и в частности гипермедиа-ориентированные скрипты в контексте гипермедиа-управляемого приложения.

Допустимо ли использовать скрипты?

Веб часто критикуют за неправильное использование. Существует распространенное убеждение, что Всемирная паутина создавалась как система доставки «документов» и стала использоваться для «приложений» только случайно или в результате странного стечения обстоятельств.

Тем не менее концепция гипермедиа ставит под вопрос четкое разделение документов и приложений. Такие системы гипермедиа, как HyperCard (предшественник веб-среды), предоставляли богатые возможности для активных и интерактивных взаимодействий, включая поддержку скриптов.

HTML в том виде, в котором он определен в спецификации и реализован, не обладает средствами для построения высокоинтерактивных приложений. Однако это не означает, что системы гипермедиа *предназначены* для «документов», а не для «приложений».

Несмотря на теоретические обоснования, дело скорее в недоработке реализации. Так как JavaScript оставался единственной точкой расширения, а гипермедиа-элементы управления не были достаточно интегрированы с JavaScript

(почему нельзя щелкнуть по ссылке без остановки программы?), разработчики не освоили гипермедиа на внутреннем уровне и использовали веб-среду как примитивный канал для приложений, имитирующих «нативные» инструменты.

В этой книге мы стараемся показать, что разработчики могут строить современные веб-приложения с использованием родоначальной технологии веб-среды — гипермедиа. При этом разработчикам приложений не приходится прибегать к абстракциям, предоставляемым большими популярными фреймворками JavaScript.

Конечно, сама библиотека `htmx` написана на JavaScript, и одно из ее преимуществ заключается в том, что взаимодействия гипермедиа, проходящие через `htmx`, предоставляют богатый интерфейс для кода JavaScript с конфигурацией, событиями и собственной поддержкой расширений `htmx`.

`Htmx` расширяет выразительность HTML до такой степени, что во многих случаях отпадает необходимость в скриптах. Это делет библиотеку `htmx` привлекательной для тех, кто не хочет писать JavaScript, а таких разработчиков, знакомых со сложностью фреймворков SPA, немало.

Тем не менее проект `htmx` не стремится отказаться от JavaScript. Целью `htmx` является не отказ от JavaScript, а написание меньшего объема кода, который будет более удобочитаемым и гипермедиа-ориентированным.

Скрипты значительно расширяют возможности веб-среды. Используя скрипты, разработчики веб-приложений могут не только усовершенствовать веб-сайты на основе HTML, но и создавать полнофункциональные приложения на стороне клиента, которые часто успешно конкурируют с нативными, толстыми клиентами.

JavaScript-центричный подход к построению веб-приложений свидетельствует о богатых возможностях веб-технологий и веб-браузеров в частности. Он занимает свою нишу в веб-разработке: в некоторых ситуациях решения гипермедиа просто не могут обеспечить такой же уровень взаимодействия, как SPA.

Тем не менее в дополнение к JavaScript-центричному стилю мы стремимся создать стиль скриптового программирования, более совместимый с гипермедиа-управляемыми приложениями и соответствующий им.

Скрипты для гипермедиа

Позаимствовав у Роя Филдинга концепцию «ограничений», определяющих REST, мы предложим два ограничения для гипермедиа-ориентированных

скриптов. Скрипты можно считать HDA-совместимыми, если соблюдаются следующие два ограничения:

- основным форматом данных, передаваемых между сервером и клиентом, остается гипермедиа — так же, как и без скриптов;
- состояние на стороне клиента за пределами модели DOM сведено к минимуму.

Эти ограничения сужают применение скриптов до той области, в которой они проявляют себя лучше всего и где другие технологии даже не могут с ними конкурировать: *проектирование взаимодействий*. Бизнес-логика и логика представления относятся к ответственности сервера, для которого можно выбрать любые языки или инструменты, подходящие для предметной области.

СЕРВЕР

Размещение бизнес-логики и логики представления «на сервере» не означает, что эти две «зоны ответственности» смешиваются друг с другом. Они могут представлять собой модули на сервере. Собственно, они *должны* представлять собой модули на сервере, как и все остальные зоны ответственности приложения.

Также заметим, что за скромным понятием «сервер» в терминологии веб-разработки обычно скрывается целый комплекс аппаратных стоек, виртуальных машин, контейнеров и т. д. Даже всемирные сети вычислительных центров сокращаются до слова «сервер» при обсуждении серверной стороны приложений HDA.

Для соблюдения этих двух ограничений иногда приходится отклоняться от того, что обычно считается лучшей практикой для JavaScript. Следует учитывать, что культурный контекст JavaScript в основном разрабатывался в JavaScript-центричных приложениях SPA.

Приложения HDA не могут с такой же простотой положиться на традиции. Считайте эту главу нашим вкладом в разработку нового стиля и передовых практик для того, что мы называем гипермедиа-управляемыми приложениями (HDA).

К сожалению, простое перечисление «передовых практик» редко выглядит убедительно. Откровенно говоря, оно скучно.

Поэтому мы продемонстрируем эти передовые практики на примере реализации клиентской функциональности в Contact.app. Чтобы рассмотреть разные свойства гипермедиа-ориентированных скриптов, мы реализуем три разные возможности.

- Раскрывающееся меню для действий *Edit (Редактировать)*, *View (Просмотр)* и *Delete (Удалить)*, устраняющее визуальную перегруженность списка контактов.
- Улучшенный интерфейс группового удаления.
- Комбинацию клавиш для передачи фокуса полю поиска.

В реализации каждой из этих возможностей важно то, что, хотя они реализуются полностью на стороне клиента с использованием скриптов, они не обмениваются информацией с сервером в формате, не относящемся к гипермедиа (например, JSON), и не хранят значительное состояние за пределами модели DOM.

Средства написания скриптов для веба

Основным языком скриптов для веба является, конечно, JavaScript. В современной веб-разработке он используется повсеместно.

Немного интересных фактов из истории интернета: JavaScript не всегда был единственным встроенным вариантом. Как намекает цитата Роя Филдинга, приведенная в начале главы, «апплеты», написанные на других языках (например, Java), также считались частью скриптовой инфраструктуры веба. Кроме того, был период, когда в Internet Explorer поддерживался VBScript — скриптовый язык, основанный на Visual Basic.

В наши дни существует множество разнообразных *транскомпиляторов* (часто сокращаемых до *транспиляторов*), преобразующих на JavaScript многие другие языки: TypeScript, Dart, Kotlin, ClojureScript, F# и т. д. Также существует формат байт-кода WebAssembly (WASM), который поддерживается в качестве цели компиляции для C, Rust и WASM-ориентированного языка AssemblyScript.

Однако большинство таких решений не адаптировано для гипермедиа-ориентированного стиля написания скриптов. Языки компиляции в JS часто используются в сочетании со SPA-ориентированными библиотеками (Dart и AngularDart, ClojureScript и Reagent, F# и Elm), а WASM в настоящее время в основном используется для компоновки с библиотеками C/C++ из JavaScript.

Мы сосредоточимся на трех скриптовых технологиях на стороне клиента, которые можно назвать гипермедиа-ориентированными.

- Vanilla JS, то есть базовый, или «ванильный», JavaScript без зависимости от каких-либо фреймворков.
- Alpine.js, библиотека JavaScript для добавления поведения прямо в HTML.
- _hyperscript — скриптовый язык, созданный вместе с `htmx`. Как и AlpineJS, _hyperscript обычно встраивается в HTML.

Рассмотрим кратко каждый из них, чтобы вы знали, с чем имеете дело.

Как и в случае с CSS, мы расскажем о каждом варианте ровно столько, сколько необходимо, чтобы показать, как он работает, и — хочется надеяться — вызвать у вас интерес для более подробного изучения.

Ванильный JS

> “ Не существует кода быстрее, чем несуществующий код.
>
> *Merb*

Под «ванильным» JavaScript понимается простой JavaScript без промежуточных уровней. Определение «ванильный» вошло в жаргон фронтенд-разработки, так как разработчики стали думать, что любое достаточно «продвинутое» веб-приложение должно использовать какую-нибудь библиотеку с названием, после которого идет суффикс «.js». По мере того как JavaScript набирал зрелость как скриптовый язык, стандартизированный между браузерами и предоставлявший все больше функциональности, эти фреймворки и библиотеки стали играть менее важную роль.

Как ни парадоксально, по мере того как JavaScript становился более мощным и отпала необходимость в первом поколении библиотек JavaScript (таких, как jQuery), люди смогли писать сложные библиотеки SPA. Библиотеки SPA часто оказываются даже более сложными, чем первое поколение библиотек JavaScript.

Следующая цитата с сайта http://vanilla-js.com (который стоит посетить, хотя он слегка устарел) отлично иллюстрирует ситуацию:

> “ VanillaJS — самый разносторонний и эффективный фреймворк, которым мне доводилось пользоваться.
>
> *http://vanilla-js.com*

С выходом JavaScript на позицию зрелого скриптового языка это стало справедливо для многих приложений. Особенно в случае HDA, так как при использовании гипермедиа приложению не понадобятся многие возможности, обычно предоставляемые сложными фреймворками SPA на основе JavaScript:

- маршрутизация на стороне клиента;
- абстракция для манипуляций с DOM (то есть шаблоны, которые автоматически обновляются при изменении используемой в них переменной);

- рендеринг на стороне сервера[1];
- присоединение динамического поведения к тегам, генерируемым сервером при загрузке (так называемое наполнение, или гидратация);
- сетевые запросы.

Без обработки всех этих сложностей в JavaScript потребности во фреймворке радикально сокращаются.

Одна из самых замечательных особенностей «базового» JS — простота установки: ее вообще нет! Вы начинаете писать JavaScript в веб-приложении, и все просто работает.

Это были хорошие новости. Есть и плохие: несмотря на усовершенствования за последнее десятилетие, JavaScript все еще обладает рядом ограничений как скриптовый язык и может оказаться далеко не лучшей автономной скриптовой технологией для приложений HDA.

- При такой богатой истории он оброс множеством возможностей и мелких недостатков.
- Он содержит сложный и запутанный набор средств для работы с асинхронным кодом.
- Работать с событиями неожиданно трудно.
- DOM API (большая часть которых изначально проектировалась для Java — да, именно *Java*) получились слишком объемными. А еще они не повышают удобство работы с часто используемой функциональностью.

Конечно, ни одно из этих ограничений не является критичным. Многие из них постепенно сходят на нет, и все больше разработчиков предпочитают «низкоуровневую» (за неимением лучшего термина) природу «базового» JavaScript более сложным решениям со скриптами на стороне клиента.

Простой счетчик

Чтобы наглядно представить использование ванильного JavaScript для фронтенд-скриптов, создадим простой виджет-счетчик.

Виджеты-счетчики — общепринятый аналог «Hello World» для фреймворков JavaScript. Будет полезно увидеть, как они могут быть реализованы в ванильном JavaScript (а также в других технологиях, которые мы также рассмотрим).

---

[1] Под «рендерингом» в данном случае понимается генерирование HTML. В приложениях HDA рендеринг на стороне сервера не требует поддержки фреймворка, поскольку генерирование HTML на сервере происходит по умолчанию.

Наш виджет будет очень простым: в нем будет содержаться число, выводимое в текстовом виде, и кнопка для увеличения этого числа.

Одна из проблем при решении подобных задач в ванильном JavaScript, в отличие от большинства фреймворков, — отсутствие правил написания кода и архитектуры по умолчанию.

В ванильном JavaScript нет никаких правил!

И это не всегда плохо. Отсутствие правил предоставляет отличную возможность ознакомиться с разными стилями JavaScript.

Встроенная реализация

Для начала рассмотрим самое простое решение, которое только можно себе представить: весь код JavaScript встраивается прямо в HTML. По щелчку на кнопке виджет ищет элемент `output` и увеличивает содержащееся в нем число.

Листинг 143. Счетчик в ванильном JavaScript (встроенная версия)

```
<section class="counter">
 <output id="my-output">0</output> ❶
 <button
  onclick=" ❷
   document.querySelector('#my-output') ❸
    .textContent++ ❹
  "
 >Increment</button>
</section>
```

❶ Элемент output снабжается идентификатором для упрощения поиска.
❷ Атрибут onclick используется для добавления прослушивателя события.
❸ Поиск output вызовом querySelector().
❹ JavaScript позволяет использовать оператор ++ со строками.

В целом неплохо.

Да, это не самый красивый код, и он может вам не нравиться, особенно если вы не привыкли работать с DOM API.

Немного раздражает необходимость добавлять идентификатор в элемент `output`. Функция `document.querySelector()` кажется слишком длинной по сравнению, скажем, с функцией `$`, предоставляемой jQuery.

Тем не менее решение работает. Его несложно понять, и очень важно, что оно не требует других библиотек JavaScript.

Это простое, встроенное решение, использующее ванильный JS.

Выделение скриптов

Хотя встроенная реализация в каком-то отношении проста, более стандартный подход заключается в перемещении этого кода в отдельный файл JavaScript. Файл JavaScript либо подключается с помощью тега `<script src>`, либо включается во встроенный тег `<script>` процессом сборки.

В этом решении HTML и JavaScript *отделяются* друг от друга и хранятся в разных файлах. Разметка HTML становится «чище» в том смысле, что в ней отсутствует код JavaScript.

Код JavaScript получается чуть более сложным, чем во встроенной версии: нам приходится искать кнопку с использованием селектора запроса и добавлять *прослушиватель события* для обработки события щелчка и увеличения счетчика.

Листинг 144. Разметка HTML для счетчика

```
<section class="counter">
 <output id="my-output">0</output>
 <button class="increment-btn">Increment</button>
</section>
```

Листинг 145. Код JavaScript для счетчика

```
const counterOutput = document.querySelector("#my-output") ❶
const incrementBtn = document.querySelector(".counter .increment-btn") ❷

incrementBtn.addEventListener("click", e => { ❸
 counterOutput.innerHTML++ ❹
})
```

❶ Находит элемент output.

❷ Находит кнопку.

❸ Используется функция addEventListener, которая по многим причинам предпочтительнее onclick.

❹ Логика остается прежней, изменяется только окружающая ее структура.

Перемещая JavaScript в другой файл, мы следуем принципу программного проектирования, известному как *«разделение ответственности»*, или *SoC* (Separation of Concerns).

Принцип разделения ответственности гласит, что разные «ответственности» (или функции) программного проекта должны находиться в разных файлах, чтобы они не «загрязняли» друг друга. JavaScript не разметка, поэтому он должен размещаться не в HTML, а *где-то в другом месте*. Точно так же информация стилей не является разметкой, поэтому она также должна храниться в отдельном файле (файле CSS, например).

Довольно долгое время принцип разделения ответственности считался «ортодоксальным» подходом к построению веб-приложений. Заявленная цель принципа —

возможность изменения и эволюции каждой ответственности независимо от других, с уверенностью в том, что это не нарушит другие ответственности.

Посмотрим, как этот принцип сработал в нашем простом примере со счетчиком. Если присмотреться к новой разметке HTML, мы видим, что к кнопке пришлось добавить класс. Это было сделано для того, чтобы кнопку можно было найти в JavaScript, а также добавить обработчик для события `click`.

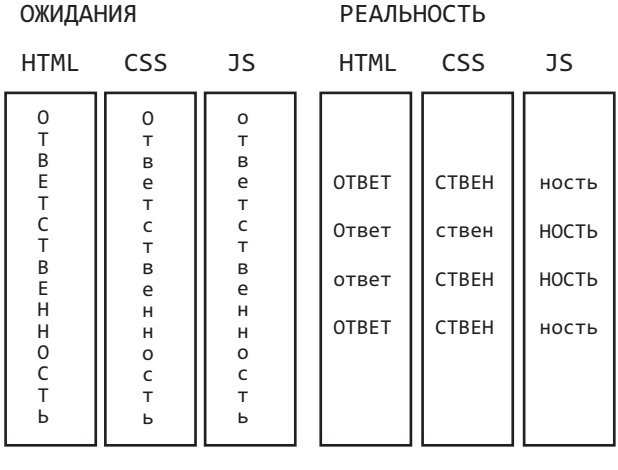
Как в HTML, так и в JavaScript имя класса представляет собой обычную строку, и нет никакого процесса, *проверяющего*, что кнопке или ее родительским элементам присвоены верные классы (что гарантировало бы, что обработчик события добавлен к нужному элементу).

К сожалению, оказалось, что невнимательное применение селекторов CSS в JavaScript может создать ситуацию, которая называется *кашей из jQuery*. Так называется ситуация, когда:

- код JavaScript, присоединяющий заданное поведение к заданному элементу, трудно найти;
- повторное использование кода затруднено;
- в итоге код оказывается хаотичным и «плоским», и в нем смешиваются разные разработчики событий, никак не связанные друг с другом.

Название «каша из jQuery» возникло потому, что многие приложения, активно использующие JavaScript, когда-то строились в jQuery (и многие продолжают строиться), а этот фреймворк поощрял (вероятно, непреднамеренно) такой стиль JavaScript.

Итак, мы видим, что концепция разделения ответственности не всегда работает так, как ожидается: ответственности переплетаются, или между ними возникают достаточно глубокие связи, даже если разделить их по разным файлам.



Чтобы показать, что проблемы разделения ответственности не ограничиваются именованием, рассмотрим еще одно небольшое изменение в HTML, также демонстрирующее проблему разделения ответственности: представьте, что вы решили преобразовать числовое поле: заменить тег `<output>` на тег `<input type="number">`.

Небольшое изменение в HTML нарушает работоспособность JavaScript, несмотря на то что мы «разделили» ответственности.

Проблема решается достаточно просто: необходимо заменить свойство `.textContent` свойством `.value`, но этот пример наглядно демонстрирует бремя синхронизации изменений разметки с кодом в разных файлах. Задача синхронизации неуклонно усложняется с ростом приложения.

Тот факт, что небольшие изменения HTML могут нарушить работу скриптов, показывает, что между разметкой и кодом существует *сильная связанность*, хотя они и разделены по разным файлам. Эта сильная связанность наводит на мысль о том, что разделение HTML и JavaScript (и CSS) часто оказывается иллюзорным: ответственности настолько связаны друг с другом, что разделить их практически нереально.

В Contact.app нас *не интересует* «структура», «стили» или «поведение»; важно лишь собрать информацию о контактах и представить ее пользователям. Принцип разделения ответственности в том варианте, в каком он сформулирован в ортодоксальной веб-разработке, в действительности не нерушимое архитектурное правило, а скорее стилистический выбор, который, как вы видите, иногда даже мешает.

Локальность поведения

Оказывается, существует целое движение, *сопротивляющееся* принципу разделения ответственности. Возьмем следующие веб-технологии и средства:

- JSX;
- LitHTML;
- CSS-in-JS;
- однофайловые компоненты;
- маршрутизация на основе файловой системы.

Каждая из этих технологий требует *совместного размещения* кода, написанного на разных языках, но реализующих одну *функциональность* (как правило, виджет UI).

Все они соединяют ответственность *реализации* для представления унифицированной абстракции конечному пользователю. Можно сказать, что раз-

деление технической ответственности — не основная, хм, ответственность разработчика.

Локальность поведения, или *LoB* (Locality of Behavior), — альтернативный принцип проектирования программных архитектур, который мы сформулировали в целях противопоставления принципу разделения ответственности. Он описывает следующие характеристики программного продукта:

> “ Для понимания поведения блока кода должно быть достаточно просто посмотреть на этот блок.
>
> https://htmx.org/essays/locality-of-behaviour/

Проще говоря, назначение кнопки должно быть понятно из кода разметки для этой кнопки. Впрочем, это не означает, что всю реализацию можно использовать как встроенную, а лишь то, что ее не придется подолгу искать или что для ее нахождения требуется знание кодовой базы.

Мы продемонстрируем локальность поведения во всех примерах — как в демонстрационных версиях счетчиков, так и в новой функциональности, добавляемой в Contact.app. Локальность поведения относится к числу явных целей проектирования как _hyperscript и Alpine.js (о чем будет рассказано позже), так и `htmx`.

Все эти инструменты обеспечивают локальность поведения встраиванием атрибутов непосредственно в HTML (вместо необходимости искать элементы в документе при помощи селекторов CSS, чтобы добавить к ним прослушиватели событий).

По нашему мнению, в гипермедиа-управляемых приложениях принцип локальности поведения зачастую важнее, чем более традиционный принцип разделения ответственности.

Что делать со счетчиком?

Итак, следует ли вернуться к атрибуту `onclick` для решения практических задач? Бесспорно, такой подход выигрывает в отношении локальности поведения, а к его дополнительным преимуществам относится то, что он встроен в HTML.

К сожалению, у атрибутов JavaScript `on*` также имеются недостатки.

- Они не поддерживают нестандартные события.
- Отсутствует удобный механизм связывания с элементом переменных с длительным сроком жизни — все переменные уничтожаются после завершения выполнения прослушивателя события.

- Если элемент существует в нескольких экземплярах, код прослушивателя придется повторять в каждом экземпляре или воспользоваться более умным механизмом, таким как делегирование событий.
- Код JavaScript, напрямую работающий с DOM, становится слишком длинным и загромождает разметку.
- Элемент не может прослушивать события других элементов.

Возьмем типичную ситуацию: имеется временный объект, и необходимо, чтобы он закрывался, когда пользователь совершает щелчок за его пределами. В такой ситуации прослушиватель должен определяться для элемента `body`, на расстоянии от фактической разметки временного объекта. Это означает, что к элементу `body` должны присоединяться прослушиватели, относящиеся к разным компонентам, никак не связанным друг с другом. Некоторые из этих компонентов даже могут не существовать на странице при ее первом рендеринге, если они добавляются динамически после рендеринга исходной страницы HTML.

Получается, что ванильный JavaScript и локальность поведения сочетаются не так хорошо, как нам хотелось бы.

Впрочем, ситуация не безнадежна: важно понимать, что принцип локальности поведения не требует, чтобы поведение *реализовалось* в точке использования, а только *вызывалось* в ней. Иначе говоря, не нужно записывать весь код в заданном элементе; достаточно ясно обозначить, что заданный элемент *вызывает* некоторый код, который может находиться где угодно.

С учетом сказанного можно усовершенствовать принцип локальности поведения с хранением кода JavaScript в отдельном файле — при условии существования разумной системы структурирования JavaScript.

RSJS

RSJS («Reasonable System for JavaScript Structure», то есть «разумная система для архитектуры JavaScript», см. https://ricostacruz.com/rsjs/) — набор правил для архитектуры JavaScript, предназначенной для «типичных сайтов, не использующих модель SPA». RSJS решает проблему отсутствия стандартного стиля программирования в ванильном JavaScript, упомянутую выше.

Правила RSJS, наиболее актуальные для виджета счетчика:

- «Используйте атрибуты `data-`» в HTML: вызов поведения посредством добавления атрибутов данных, очевидно, указывает на использование JavaScript, а не случайных классов или идентификаторов, которые могут быть по ошибке удалены или изменены.

- «Один компонент на файл»: имя файла должно соответствовать атрибуту данных, чтобы его можно было легко найти, — безусловная польза для локальности поведения.

Чтобы выполнить рекомендации RSJS, проведем реструктуризацию текущих файлов HTML и JavaScript. Сначала мы используем *атрибуты данных* (то есть атрибуты HTML, начинающиеся с `data-`, стандартный прием в HTML), чтобы показать, что HTML описывает компонент-счетчик. Затем обновим код JavaScript для использования селектора атрибута, который ищет атрибут `data-counter` как корневой элемент компонента счетчика и связывает соответствующие обработчики событий и логику. Кроме того, переработаем код, чтобы в нем использовался вызов `querySelectorAll()`, и добавим функциональность счетчика ко *всем* компонентам-счетчикам на странице. (Никогда не знаешь заранее, сколько счетчиков может понадобиться!)

Сейчас код выглядит так:

```
<section class="counter" data-counter> ❶
 <output id="my-output" data-counter-output>0</output> ❷
 <button class="increment-btn" data-counter-increment>Increment</button>
</section>
```

❶ Вызывает поведение JavaScript с атрибутом данных.
❷ Помечает важные элементы-потомки.

```
// counter.js ❶
document.querySelectorAll("[data-counter]") ❷
 .forEach(el => {
  const
  output = el.querySelector("[data-counter-output]"),
  increment = el.querySelector("[data-counter-increment]"); ❸

  increment.addEventListener("click", e => output.textContent++); ❹
 });
```

❶ Имя файла должно совпадать с атрибутом данных, чтобы его было проще найти.
❷ Получает все элементы, вызывающие это поведение.
❸ Получает любые необходимые дочерние элементы.
❹ Регистрирует обработчики событий.

Использование RSJS решает (или, по крайней мере, уменьшает) многие проблемы, отмеченные для первого, неструктурированного примера с выделением ванильного JS в отдельный файл.

- Код JS, присоединяющий поведение к заданному элементу, *понятен* (хотя только благодаря соглашению об именах).

- Код легко *использовать повторно* — вы просто создаете на странице другой компонент-счетчик, и он работает.
- Код *хорошо организован* — одно поведение на файл.

В общем и целом RSJS предоставляет хороший способ структурирования ванильного JavaScript в гипермедиа-управляемых приложениях. Таким образом, если JavaScript не взаимодействует с сервером через простой API данных JSON и не хранит большой объект внутреннего состояния за пределами DOM, этот подход полностью совместим с подходом HDA.

Реализуем функциональность в Contact.app с использованием подхода RSJS/ванильного JavaScript.

Ванильный JS в действии: раскрывающееся меню

Домашняя страница содержит ссылки Edit, View и Delete для каждого контакта в таблице. Ссылки занимают много места и визуально загромождают страницу. Чтобы исправить этот недостаток, поместим указанные действия в раскрывающееся меню, которое будет открываться нажатием кнопки.

Если вы недостаточно хорошо знаете JavaScript и приведенный код покажется вам слишком сложным, не волнуйтесь; примеры для Alpine.js и _hyperscript, которые мы рассмотрим ниже, будут более понятными.

Начнем с общей схемы разметки, которая понадобится для раскрывающегося меню. Для начала нужно создать элемент (мы будем использовать `<div>`), в который заключается весь виджет, и пометить его как компонент меню. В этом `div` будет содержаться стандартная кнопка `<button>`, которая будет показывать и скрывать меню с командами. Наконец, понадобится еще один элемент `<div>` для команд меню, которые в нем будут выводиться.

Команды меню представляют собой простые якорные теги, как и в текущей версии таблицы контактов.

Обновленная разметка HTML, структурированная по правилам RSJS, выглядит так:

```
<div data-overflow-menu> ❶
  <button type="button" aria-haspopup="menu"
    aria-controls="contact-menu-{{ contact.id }}"
    >Options</button> ❷
  <div role="menu" hidden id="contact-menu-{{ contact.id }}"> ❸
    <a role="menuitem" href="/contacts/{{ contact.id }}/edit">Edit</a> ❹
    <a role="menuitem" href="/contacts/{{ contact.id }}">View</a>
    <!-- ... -->
  </div>
</div>
```

❶ Отмечает корневой элемент компонента меню.
❷ Кнопка открывает и закрывает меню.
❸ Контейнер для команд меню.
❹ Команды меню.

Роли и атрибуты ARIA основаны на паттернах Menu и Menu Button из руководства ARIA Authoring Practices Guide.

ЧТО ТАКОЕ ARIA?

По мере того как веб-разработчики создают все более интерактивные сайты, похожие на приложения, набора элементов HTML становится недостаточно. Как вы уже видели, при помощи CSS и JavaScript можно наделить существующие элементы расширенным поведением и оформлением, не уступающим нативным элементам.

Однако существует одна особенность, которую не могут повторить веб-приложения. Хотя содержащиеся в них виджеты могут быть *похожими* на настоящие, технологии доступности (например, экранные дикторы) могут работать только с «родными» элементами HTML.

Даже если вы не пожалеете времени и правильно настроите все взаимодействия с клавиатурой, у некоторых пользователей будут возникать проблемы с нестандартными элементами.

Спецификация ARIA была создана инициативной группой WAI (Web Accessibility Initiative) комитета W3C для решения указанной проблемы. Если говорить упрощенно, спецификация определяет набор атрибутов, которые можно добавить в разметку HTML, чтобы она воспринималась такими программными средствами доступности, как экранные дикторы.

ARIA определяет два главных компонента, которые взаимодействуют друг с другом. Первый – атрибут `role`. Этот атрибут имеет заранее определенный набор допустимых значений: `menu`, `dialog`, `radiogroup` и т. д. Атрибут `role` не добавляет никакое поведение в элементы HTML. Скорее это обещание, которое вы даете пользователю. Помечая элемент атрибутом `role='menu'`, вы тем самым говорите: *«Я сделаю так, что этот элемент будет работать как меню»*.

Если вы добавите роль к элементу, но не выполните свое обещание, для многих пользователей опыт взаимодействия будет хуже, чем если бы для элемента атрибут `role` вообще не определялся. Поэтому:

> “ Не поддерживать ARIA вообще — лучше, чем поддерживать ARIA плохо.
>
> *W3C Read Me First | APG*
> https://www.w3.org/WAI/ARIA/apg/practices/read-me-first/

Вторым компонентом ARIA являются *состояния и свойства* с общим префиксом `aria-`: `aria-expanded`, `aria-controls`, `aria-label` и т. д. Этими атрибутами могут определяться разные особенности: состояние виджета, отношения между компонентами или дополнительная семантика. Как и в предыдущем случае, эти атрибуты являются обещаниями, а не реализациями.

Вместо того чтобы изучать все роли и атрибуты и пытаться объединить их в полезный виджет, лучше положиться на руководство APG (ARIA Authoring Practices Guide) – веб-ресурс с практической информацией, предназначенной непосредственно для веб-разработчиков.

Если у вас нет опыта использования ARIA, обратитесь к следующим ресурсам W3C:

- ARIA: Read Me First: https://www.w3.org/WAI/ARIA/apg/practices/read-me-first/
- ARIA UI patterns: https://www.w3.org/WAI/ARIA/apg/patterns/
- ARIA Good Practices: https://www.w3.org/WAI/ARIA/apg/practices/

Никогда не забывайте *тестировать* создаваемый сайт на доступность, чтобы все пользователи могли работать с ним легко и эффективно.

После краткого знакомства с ARIA вернемся к раскрывающемуся меню. Начнем с шаблонной схемы RSJS: запрос ко всем элементам с некоторым атрибутом данных, перебор этих элементов и получение всех актуальных потомков.

Обратите внимание: в приведенном ниже коде мы несколько изменили шаблонную схему RSJS для интеграции с `htmx`; раскрывающееся меню будет загружаться при загрузке нового контента в `htmx`.

```
function overflowMenu(subtree = document) {
 document.querySelectorAll("[data-overflow-menu]").forEach(menuRoot => { ❶
  const
  button = menuRoot.querySelector("[aria-haspopup]"), ❷
  menu = menuRoot.querySelector("[role=menu]"), ❷
  items = [...menu.querySelectorAll("[role=menuitem]")]; ❸
 });
}

addEventListener("htmx:load", e => overflowMenu(e.target)); ❹
```

❶ С RSJS часто приходится использовать конструкцию document.querySelectorAll(...).forEach.

❷ Чтобы сохранить чистоту разметки HTML, мы используем атрибуты ARIA вместо нестандартных атрибутов данных.

❸ Применяем оператор расширения (spread) для преобразования NodeList в обычный Array.

❹ Все раскрывающиеся меню инициализируются при загрузке страницы или вставке контента htmx.

Традиционно для хранения информации о том, открыто ли меню, использовалась переменная JavaScript или свойство объекта состояния JavaScript. Такой подход часто встречается в крупных веб-приложениях с большим объемом JavaScript.

Тем не менее у такого подхода есть свои недостатки:

- необходимость обеспечения синхронизации DOM с состоянием (что сложнее делать без фреймворка);
- потеря возможности сериализации HTML (так как открытое состояние хранится не в DOM, а в JavaScript).

Вместо того чтобы выбирать этот путь, воспользуемся DOM для хранения состояния. Для проверки того, находится ли меню в закрытом состоянии, будет использоваться атрибут `hidden`. Если сохранить разметку HTML страницы и перезагрузить ее, меню может быть восстановлено простым повторным выполнением JS.

```
 items = [...menu.querySelectorAll("[role=menuitem]")];

 const isOpen = () => !menu.hidden; ❶

});
```

❶ Атрибут hidden воспроизводится как *свойство* hidden, благодаря чему нам не придется использовать getAttribute.

Для команд меню также стоит отключить переход по клавише Tab, чтобы управлять их фокусом самостоятельно.

```
 const isOpen = () => !menu.hidden; ❶

 items.forEach(item => item.setAttribute("tabindex", "-1"));

});
```

Реализуем переключение состояния меню в JavaScript:

```
 items.forEach(item => item.setAttribute("tabindex", "-1"));

 function toggleMenu(open = !isOpen()) { ❶
  if (open) {
   menu.hidden = false;
   button.setAttribute("aria-expanded", "true");
   items[0].focus(); ❷
```

```
  } else {
   menu.hidden = true;
   button.setAttribute("aria-expanded", "false");
  }
 }
 toggleMenu(isOpen()); ❸
 button.addEventListener("click", () => toggleMenu()); ❹
 menuRoot.addEventListener("blur", e => toggleMenu(false)); ❺

})
```

❶ Необязательный параметр для определения желаемого состояния. Это позволяет использовать одну функцию для открытия, закрытия или переключения состояния меню.
❷ Передает фокус первой команде меню при его открытии.
❸ Вызывает toggleMenu с текущим состоянием для инициализации атрибутов элемента.
❹ Переключает состояние меню по щелчку кнопки.
❺ Закрывает меню при потере фокуса.

Кроме того, сделаем так, чтобы меню закрывалось по щелчку за его границами, — удобное поведение, имитирующее работу нативных раскрывающихся меню. Эта функциональность потребует назначения прослушивателя события для всего окна.

Заметим, что с такими прослушивателями событий нужно быть осторожными: может оказаться, что добавляемые компонентами прослушиватели накапливаются и не удаляются при удалении компонента из DOM. К сожалению, это приводит к утечкам памяти, которые довольно трудно обнаружить.

В JavaScript нет простого способа выполнения логики при удалении элемента. Лучшим вариантом остается так называемый `MutationObserver` API (API наблюдателей за изменениями). Объект `MutationObserver` очень полезен, но API тяжеловесен и не отличается простотой, поэтому в нашем примере он использоваться не будет.

Вместо этого мы воспользуемся простым паттерном для предотвращения утечек прослушивателей событий. При выполнении прослушивателя мы проверим, остается ли присоединяющий компонент в DOM, и если компонент уже отсутствует, удалим прослушиватель и вернем управление.

Это довольно примитивная ручная реализация *сборки мусора*. Как это (обычно) бывает с алгоритмами сборки мусора, наша стратегия удаляет прослушиватели по прошествии недетерминированного времени после того, как надобность в них отпадет. К счастью, при частых событиях типа «пользователь щелкает в любом месте на странице», управляющих сборкой мусора, она должна работать достаточно хорошо.

```
 menuRoot.addEventListener("blur", e => toggleMenu(false));

 window.addEventListener("click", function clickAway(event) {
  if (!menuRoot.isConnected) window.removeEventListener("click",
clickAway); ❶
  if (!menuRoot.contains(event.target)) toggleMenu(false); ❷
 });
});
```

❶ В этой строке выполняется сборка мусора.

❷ Если щелчок выполнен за пределами меню, то меню закрывается.

Перейдем к клавиатурным взаимодействиям для раскрывающегося меню. Обработчики событий клавиатуры похожи друг на друга и не отличаются сложностью, поэтому приведем их в одном блоке.

```
  if (!menuRoot.contains(event.target)) toggleMenu(false); ❷
 });

 const currentIndex = () => { ❶
  const idx = items.indexOf(document.activeElement);
  if (idx === -1) return 0;
  return idx;
 }

 menu.addEventListener("keydown", e => {
  if (e.key === "ArrowUp") {
   items[currentIndex() - 1]?.focus(); ❷

  } else if (e.key === "ArrowDown") {
   items[currentIndex() + 1]?.focus(); ❸

  } else if (e.key === "Space") {
   items[currentIndex()].click(); ❹

  } else if (e.key === "Home") {
   items[0].focus(); ❺

  } else if (e.key === "End") {
   items[items.length - 1].focus(); ❻

  } else if (e.key === "Escape") {
   toggleMenu(false); ❼
   button.focus(); ❽
  }
 });
});
```

❶ Функция-хелпер — получает индекс в массиве элементов для команды меню, обладающей фокусом (0, если таких команд нет).
❷ Передает фокус предыдущей команде меню при нажатии клавиши со стрелкой вверх.
❸ Передает фокус следующей команде меню при нажатии клавиши со стрелкой вниз.
❹ Активирует команду меню, обладающую фокусом, при нажатии клавиши «пробел».
❺ Передает фокус первой команде меню при нажатии клавиши Home.
❻ Передает фокус последней команде меню при нажатии клавиши End.
❼ Закрывает меню при нажатии клавиши Escape.
❽ Возвращает фокус кнопке меню при закрытии меню.

Этот код делает все, что требуется. Да, он получился довольно длинным. Но в нем реализован действительно большой объем поведения.

Пока что меню неидеально, и в нем отсутствуют многие возможности. Например, мы не поддерживаем подменю или динамическое добавление/удаление команд меню. Если нам понадобится более серьезная функциональность такого рода, лучше воспользоваться готовой библиотекой, например `details-menu-element`[1] с GitHub.

Но для нашего относительно простого сценария вполне достаточно ванильного JavaScript, а в ходе работы над реализацией вы еще попутно познакомились с ARIA и RSJS.

Alpine.js

Итак, вы узнали, как структурировать простой код JavaScript в стиле VanillaJS. А теперь обратимся к реальному фреймворку JavaScript, который позволяет использовать другой подход для добавления динамического поведения в приложение, — Alpine.js[2].

Alpine — относительно новая библиотека JavaScript. С ее помощью разработчики могут внедрять код JavaScript прямо в HTML по аналогии с атрибутами `on*`, доступными в простом HTML и JavaScript. Тем не менее Alpine поднимает концепцию внедрения скриптов на более высокий уровень, чем атрибуты `on*`.

Alpine характеризует себя как современную замену jQuery — широко используемой, более старой библиотеки JavaScript. Как вы увидите, это описание вполне соответствует действительности.

---

[1] https://github.com/github/details-menu-element

[2] https://alpinejs.dev/

Установить Alpine очень просто: это единственный файл, свободный от зависимостей, который можно просто подключить через CDN.

Листинг 146. Установка Alpine

```
<script src="https://unpkg.com/alpinejs"></script>
```

Также можно выполнить установку через менеджер пакетов (такой, как NPM) или загрузить с вашего собственного сервера.

Alpine предоставляет набор атрибутов HTML, имена которых начинаются с префикса `x-`; главным из них является `x-data`. Содержимое `x-data` представляет собой выражение JavaScript, результатом вычисления которого является объект. К свойствам этого объекта можно обратиться из элемента, в котором находится атрибут `x-data`.

Чтобы получить представление об AlpineJS, посмотрим, как с этим фреймворком реализуется наш пример со счетчиком.

Единственное состояние, которое нужно хранить для счетчика, — текущее значение. Объявим объект JavaScript с одним свойством, `count`, в атрибуте `x-data` элемента `div` счетчика.

Листинг 147. Счетчик на основе Alpine, строка 1

```
<div class="counter" x-data="{ count: 0 }">
```

Таким образом определяется состояние, то есть данные, которые будут использоваться для управления динамическими обновлениями DOM. При таком объявлении состояния мы можем использовать его *внутри* элемента `div`, в котором оно обновляется. Добавим элемент `output` с атрибутом `x-text`.

Затем мы *свяжем* атрибут `x-text` с атрибутом `count`, объявленным в атрибуте `x-data` родительского элемента `div`. Это приводит к тому, что текстом элемента `output` становится текущее значение `count`: при обновлении `count` также будет обновляться текст `output`. Такая модель программирования называется «реактивной», потому что DOM «реагирует» на изменения в поддерживающих данных.

Листинг 148. Счетчик на основе Alpine, строка 2

```
<div x-data="{ count: 0 }">
 <output x-text="count"></output> ❶
```

❶ Атрибут x-text.

Затем необходимо обновить счетчик с использованием кнопки. Alpine позволяет присоединять обработчики событий при помощи атрибута `x-on`.

Чтобы назначить событие для прослушивания, поставьте после имени атрибута `x-on` двоеточие, а затем укажите имя события. Значением атрибута является выполняемый код JavaScript. В этом атрибут `x-on` похож на простые атрибуты `on*`, рассмотренные выше, но обладает существенно большей гибкостью.

Для счетчика нужно прослушивать событие `click` и увеличивать счетчик по щелчку. Код Alpine будет выглядеть так:

Листинг 149. Счетчик на основе Alpine, полный код

```
<div x-data="{ count: 0 }">
 <output x-text="count"></output>

 <button x-on:click="count++">Increment</button> ❶
</div>
```

❶ С x-on событие задается в имени атрибута.

Собственно, это все. Простой компонент, такой как счетчик, должно быть легко реализовать, и Alpine соответствует этому условию.

x-on:click и onclick

Как уже говорилось, атрибут Alpine `x-on:click` (или его сокращенная запись, атрибут `@click`) похож на встроенный атрибут `onclick`. Однако у него есть ряд дополнительных функций, которые делают его намного более полезным.

- Можно прослушивать события других элементов. Например, модификатор `.outside` позволяет прослушивать любые события `click`, происходящие *не* внутри элемента.
- Другие модификаторы могут использоваться для:
  - регулировки и устранения дребезга событий;
  - игнорирования событий, всплывающих от элементов-потомков;
  - присоединения пассивных слушателей.
- Можно прослушивать нестандартные события. Например, для прослушивания события `htmx:after-request` используется запись `x-on:htmx:after-request="doSomething()"`.

Реактивность и шаблоны

Надеемся, вы согласитесь с тем, что версия виджета счетчика для AlpineJS в целом лучше реализации с ванильным JS, которая выглядела громоздко или занимала несколько файлов.

Большая часть возможностей AlpineJS связана с поддержкой концепции «реактивных» переменных, позволяющих связать счетчик элемента `div` с переменной, к которой можно обращаться как из `output`, так и из `button`, и правильным обновлением всех зависимостей при возникновении изменения. Alpine поддерживает намного более сложные привязки данных, чем были показаны выше, и это отличная скриптовая библиотека общего назначения на стороне клиента.

Alpine.js в действии: панель инструментов для групповых действий

Реализуем функциональность Contact.app с Alpine. В текущей версии Contact.app у нижнего края страницы находится кнопка Delete Selected Contacts (Удалить выбранные контакты). У кнопки длинное имя, ее непросто найти, и она занимает много места. Если потребуется добавить дополнительные «групповые» действия, решение будет плохо масштабироваться.

В этом разделе мы заменим одну кнопку панелью инструментов. Более того, панель инструментов будет отображаться только при выделении пользователем нескольких контактов. Наконец, на панели будут выводиться количество выделенных контактов и инструменты для выделения всех контактов одной операцией.

Начнем с добавления атрибута `x-data` для хранения состояния, которое будет использоваться для проверки того, видима панель инструментов или нет. Его необходимо применить к родительскому элементу как панели инструментов, которую мы будем добавлять, так и к флажкам, состояние которых будет обновляться при установке и сбросе. С данной разметкой HTML лучше всего поместить атрибут в элемент `form`, в который заключена таблица контактов. Мы объявим свойство `selected`, в котором будет храниться массив с идентификаторами выделенных контактов на основании отмеченных флажков (чекбоксов).

Тег `form` будет выглядеть так:

```
<form x-data="{ selected: [] }"> ❶
```

❶ В форму заключена таблица контактов.

Затем в верхней части таблицы контактов добавим тег `template`. Тег `template` *не* рендерится браузером по умолчанию, и, возможно, вас удивит, что мы его используем. Тем не менее, добавляя атрибут Alpine `x-if`, можно указать Alpine: если условие истинно, выведи HTML из этого шаблона.

Вспомните, что панель инструментов должна отображаться в том и только в том случае, если в таблице выделен один или несколько контактов. Но мы знаем,

что идентификаторы выделенных контактов хранятся в свойстве `selected`. Проверка *длины* массива позволяет легко определить, присутствуют ли в таблице выделенные контакты.

```
<template x-if="selected.length > 0"> ❶
 <div class="box info tool-bar">
  <slot x-text="selected.length"></slot>
  contacts selected

  <button type="button" class="bad bg color border">Delete</button> ❷
  <hr aria-orientation="vertical">
  <button type="button">Cancel</button>
 </div>
</template>
```

❶ Эта разметка HTML отображается, если пользователь выделил в таблице 1 или несколько контактов.

❷ Вскоре мы реализуем эти кнопки.

На следующем шаге необходимо проверить, что при изменении состояния флажка для некоторого контакта идентификатор этого контакта добавляется в свойство `selected` (или удаляется из него). Для этого необходимо использовать новый атрибут Alpine `x-model`. Атрибут `x-model` *связывает* заданный элемент с некоторыми нижележащими данными, то есть с его «моделью».

В данном случае требуется связать значение полей ввода `checkbox` со свойством `selected`. Делается это так:

```
<td>
<input type="checkbox" name="selected_contact_ids" value="{{ contact.id }}"
x-model="selected"> ❶
</td>
```

❶ Атрибут x-model связывает значение этого поля ввода со свойством selected.

Теперь при установке или сбросе флажка массив `selected` будет обновлен идентификатором контакта заданной строки данных. Более того, изменения, вносимые в массив `selected`, будут аналогичным образом отражены в состоянии флажков. Это называется *двусторонним* связыванием.

С этим кодом панель инструментов будет появляться и исчезать в зависимости от того, есть ли в таблице выделенные флажки контактов.

Очень удобно.

Прежде чем двигаться дальше, обратите внимание, что в коде встречаются ссылки `class=`. Они нужны для стилевого оформления CSS и не являются частью Alpine.js. Мы включили их только как напоминание о том, что для нормальной

работы создаваемого меню потребуется CSS. Классы в приведенном выше коде относятся к минимальной библиотеке CSS, которая называется `Missing.css`. Если вы используете другие библиотеки CSS, такие как Bootstrap, Tailwind, Bulma, Pico.css и т. д., ваш код стилей будет выглядеть иначе.

Реализация действий

Итак, теперь у нас есть механизм показа и скрытия панели инструментов; посмотрим, как реализовать кнопки внутри нее.

Начнем с реализации кнопки Clear (Очистить), потому что она проще остальных. Все, что потребуется, — очистить массив `selected` по щелчку на кнопке. Из-за двустороннего связывания, предоставляемого Alpine, при этом будут сброшены флажки всех выделенных контактов (после чего панель инструментов скрывается).

С кнопкой Cancel (Отменить) тоже все несложно:

```
<button type="button" @click="selected = []">Cancel</button> ❶
```

❶ Сбрасывает массив selected.

И снова с AlpineJS это делается очень просто.

С кнопкой Delete (Удалить), однако, сложнее. Для ее реализации необходимо решить две задачи: во-первых, запросить подтверждение того, что пользователь действительно хочет удалить все выделенные контакты. Затем, если пользователь подтвердит действие, выдать запрос `DELETE` через JavaScript API для `htmx`.

```
<button type="button" class="bad bg color border"
 @click="confirm(`Delete ${selected.length} contacts?`) && ❶
  htmx.ajax('DELETE', '/contacts', { source: $root, target: document.body
})" ❷
>Delete</button>
```

❶ Подтверждает, что пользователь желает удалить выделенные контакты.

❷ Выдает запрос DELETE с использованием JavaScript API для htmx.

Обратите внимание: мы используем поведение ускоренного вычисления оператора `&&` в JavaScript, чтобы избежать вызова `htmx.ajax()`, если вызов `confirm()` вернет `false`.

Функция `htmx.ajax()` всего лишь открывает доступ к нормальному и HTML-управляемому обмену данными гипермедиа, который открывают атрибуты HTML в `htmx` прямо из JavaScript.

Если присмотреться к вызову `htmx.ajax`, заметим: мы сначала указываем, что требуется выдать запрос `DELETE` к `/contacts`. Затем мы передаем два дополнительных блока информации: `source` и `target`. Свойство `source` определяет элемент, из которого `htmx` получит данные для включения в запрос. Мы присваиваем ему `$root` — специальное символическое имя Alpine, представляющее элемент, в котором объявляется атрибут `x-data`. В данном случае это форма, содержащая все контакты. Целевым элементом `target` (то есть элементом, в котором будет размещена разметка HTML ответа) становится тело всего документа, так как обработчик `DELETE` возвращает всю страницу при завершении.

Обратите внимание: Alpine здесь используется способом, совместимым с HDA. Можно было выдать запрос AJAX прямо из Alpine и, возможно, обновить свойство `x-data` в зависимости от его результатов. Но вместо этого мы делегируем выполнение работы JavaScript API библиотеки `htmx`, который обеспечивает *обмен гипермедиа* с сервером. Это ключевой принцип гипермедиа-ориентированного написания скриптов в приложениях HDA.

После всей подготовки выполнять групповые действия с контактами становится намного удобнее: визуальное нагромождение сокращается, и панель инструментов можно расширять новыми командами без захламления основного интерфейса приложения.

_hyperscript

Последняя скриптовая технология, которую мы рассмотрим, стоит чуть в стороне от предыдущих — это _hyperscript[1]. Авторы книги изначально создавали _hyperscript как проект, родственный `htmx`. Мы посчитали, что JavaScript был недостаточно объектно-ориентированным, что усложняло добавление мелких скриптовых улучшений в приложения `htmx`.

Хотя два предыдущих примера были ориентированы на JavaScript, _hyperscript использует совершенно иной синтаксис, основанный на более старом языке HyperTalk. HyperTalk был скриптовым языком для технологии, которая называлась HyperCard, — старой системы гипермедиа для ранних версий Macintosh.

У _hyperscript бросается в глаза одно свойство: он больше напоминает обычный текст на английском языке, чем код других языков программирования.

Как и Alpine, _hyperscript представляет собой современную замену jQuery. Так же как и Alpine, _hyperscript позволяет записывать скрипты во встроенном виде, то есть прямо в HTML.

[1] https://hyperscript.org/

Однако в отличие от Alpine _hyperscript *не*реактивен. Вместо этого _hyperscript стремится к тому, чтобы манипуляции с DOM в ответ на события было легко записывать и читать. В нем предусмотрены встроенные языковые конструкции для многих операций DOM, чтобы не приходилось возиться с перегруженными JavaScript DOM API.

Посмотрим, как выглядят скрипты в языке _hyperscript, чтобы вы могли позже заняться самостоятельным изучением языка, если он покажется вам интересным.

Как и `htmx` и AlpineJS, _hyperscript можно установить через CDN или из `npm` (имя пакета *hyperscript.org*).

Листинг 150. Установка _hyperscript из CDN

```
<script src="//unpkg.com/hyperscript.org"></script>
```

_hyperscript использует атрибут `\_` (символ подчеркивания) для включения скриптов в элементы DOM. В зависимости от потребностей в валидации HTML можно также использовать атрибуты `script` или `data-script`.

Посмотрим, как реализовать простой компонент-счетчик, который рассматривался выше, с использованием _hyperscript. Элементы `output` и `button` будут размещены внутри `div`. Чтобы реализовать счетчик, добавим в кнопку небольшой фрагмент _hyperscript. По щелчку кнопка будет увеличивать число в тексте предшествующего тега `output`.

```
<div class="counter">
 <output>0</output>
 <button _="on click increment the textContent of the previous
<output/>">Increment</button> ❶
</div>
```

❶ Код _hyperscript внедряется в button.

Рассмотрим все компоненты этого скрипта.

- `on click` — прослушиватель события, который дает команде кнопке прослушать событие `click` и выполнить следующий код.
- `increment` — «команда» _hyperscript, которая инкрементирует значения по аналогии с оператором `++` в JavaScript.
- `the` — не имеет семантического смысла в _hyperscript, но делает скрипты более удобочитаемыми.
- `textContent` — одна из форм *обращения к свойствам* в _hyperscript. Вероятно, вам знаком синтаксис JavaScript `a.b`, означающий «получить свойство `b` объ-

екта a». _hyperscript поддерживает этот синтаксис, а также формы `b of a` и `a's b`. Выбор зависит от того, какая из форм покажется вам наиболее удобочитаемой.

- `previous` — выражение _hyperscript, которое находит предыдущий элемент DOM, удовлетворяющий заданному условию.
- `<output />` — *литерал запроса*. Представляет собой селектор CSS, заключенный между символами < и />.

В этом коде ключевое слово `previous` (и парное ключевое слово `next`) — примеры того, как _hyperscript упрощает операции DOM: в стандартном DOM API подобная встроенная функциональность отсутствует, а реализовать ее в ванильном JS труднее, чем можно ожидать!

Как видите, _hyperscript чрезвычайно выразителен, особенно по части манипуляций с DOM. Это позволяет внедрять скрипты прямо в HTML: так как скриптовый язык более эффективен, написанные на нем скрипты получаются более компактными и их проще читать.

ПРОГРАММИРОВАНИЕ НА ЕСТЕСТВЕННОМ ЯЗЫКЕ?

Опытные программисты могут отнестись к _hyperscript скептически: мы знаем много проектов «программирования на естественном языке» (NLP, Natural Language Programming), предназначенных для непрограммистов и неопытных программистов. Эти проекты исходили из того, что тот, кто может прочитать код на «естественном языке», сможет его и написать. Это вело к появлению плохо написанного и структурированного кода, и в целом NLP не стоило поднятого (и часто чрезмерного) шума.

_hyperscript – это *не* язык программирования NLP. Да, его синтаксис во многих отношениях основан на речевых паттернах веб-разработчиков. Однако удобочитаемость _hyperscript достигается не за счет сложной эвристики или нечеткой обработки NLP, а скорее за счет разумного использования стандартных приемов парсинга в сочетании с удобочитаемостью.

Как видно на примере *ссылки на запрос* <output/>, _hyperscript не старается избегать использования «неестественного» языка, привязанного к DOM, там, где он требуется.

_hyperscript в действии: комбинации клавиш

Хотя демоверсия счетчика — хороший способ сравнить разные подходы к написанию скриптов, теория проверяется на практике при попытке реализовать что-нибудь полезное. В случае _hyperscript добавим в Contact.app поддержку

комбинаций клавиш: когда пользователь нажмет Alt+S в приложении, фокус будет передаваться полю поиска.

Так как комбинация передает фокус полю поиска, включим код в это поле поиска в соответствии с принципом локальности поведения.

Исходная разметка HTML для поля поиска выглядит так:

```
<input id="search" name="q" type="search" placeholder="Search Contacts">
```

Для добавления прослушивателя события используем синтаксис `on keydown`, срабатывающий при нажатии клавиши. Также в _hyperscript можно использовать синтаксис фильтра `_event` в квадратных скобках после события. В квадратных скобках размещается *выражение-фильтр*, которое отфильтровывает события нажатия клавиш, не представляющие для нас интереса. В нашем примере должны учитываться только события с удержанием клавиши Alt и нажатием клавиши «S». Можно создать логическое выражение, которое проверяет свойство `altKey` (чтобы узнать, содержит ли оно `true`) и свойство `code` (чтобы узнать, содержит ли оно `"KeyS"`).

Пока что _hyperscript выглядит так:

Листинг 151. Начало кода комбинации клавиш

```
on keydown[altKey and code is 'KeyS'] ...
```

Теперь по умолчанию _hyperscript будет прослушивать заданное событие `_on` в элементе, в котором оно было объявлено. Таким образом, с этим скриптом мы будем получать события `keydown`, если поле поиска уже имеет фокус. Но это не то, что нам нужно! Клавиши должны работать *глобально* независимо от того, какой элемент обладает фокусом.

Не проблема! Событие `keydown` можно прослушивать где угодно, для этого следует включить в обработчик события условие `from`. В данном случае событие `keyDown` должно прослушиваться в окне, а код будет выглядеть примерно так:

Листинг 152. Глобальное прослушивание

```
on keydown[altKey and code is 'KeyS'] from window ...
```

При помощи условия `from` можно присоединить прослушиватель к окну, тогда как код останется в элементе, которому он логически принадлежит.

После выбора события, которое должно использоваться для передачи фокуса полю поиска, можно реализовать передачу фокуса вызовом метода `standard .focus()`.

Ниже приведем весь код скрипта, встроенный в HTML.

Листинг 153. Окончательный код скрипта

```
<input id="search" name="q" type="search" placeholder="Search Contacts"
 _="on keydown[altKey and code is 'KeyS'] from the window
   me.focus()"> ❶
```

❶ «me» относится к элементу, в котором записан скрипт.

С учетом всей функциональности код получается довольно компактным и легкочитаемым — почти как естественный язык.

Зачем нужен еще один язык программирования?

Все это хорошо, но вы можете подумать: «Новый скриптовый язык? Это уже перебор». И в каком-то отношении будете правы: JavaScript — достойный язык программирования, он очень хорошо оптимизирован и понятен веб-разработчикам. С другой стороны, создав совершенно новый скриптовый язык для фронтенда, мы решили некоторые проблемы, из-за которых код на JavaScript становится уродливым и перегруженным.

Прозрачность асинхронных операций

В _hyperscript асинхронные функции (то есть функции, возвращающие экземпляры `Promise`) могут вызываться так, как если бы они были синхронными. Преобразование синхронной функции в асинхронную не нарушает работоспособности кода _hyperscript, из которого она вызывается. Для этого при вычислении любого выражения проверяется `Promise`, и выполняемый скрипт приостанавливается, если экземпляр существует (приостанавливается только текущий обработчик события, а основной поток не блокируется). JavaScript же требует *либо* явного использования обратных вызовов, *либо* использования явных асинхронных аннотаций (которые не могут смешиваться с синхронным кодом).

Обращения к свойствам массивов

В _hyperscript при обращении к свойству массива (кроме `length` или `number`) возвращается массив значений свойства для каждого элемента массива, благодаря чему доступ к свойствам массива работает как функция `flatMap()`. В jQuery существует похожая функциональность, но только для своей структуры данных.

Нативный синтаксис CSS

В _hyperscript такие конструкции, как классы CSS и идентификаторы-литералы, могут использоваться прямо в языке. Вам не придется обращаться с вызовами к пространному DOM API, как в JavaScript.

Глубокая поддержка событий

Работать с событиями в _hyperscript намного приятнее, чем в JavaScript, благодаря встроенной поддержке реакции на события и отправки событий, а также таким распространенным паттернам обработки ошибок, как устранение дребезга или ограничение событий по частоте. _hyperscript также предоставляет декларативные механизмы для синхронизации событий в выбранном элементе и между несколькими элементами.

И снова мы хотим подчеркнуть, что в этом примере мы не выходим за рамки HDA: в скриптах добавляется только функциональность фронтенда на стороне клиента. Мы не создаем большой объем состояния за пределами модели DOM и не управляем им, а также не осуществляем обмен данными с сервером по модели, отличной от гипермедиа.

Кроме того, поскольку _hyperscript хорошо встраивается в HTML, это помогает разработчику сосредоточиться на составляющей *гипермедиа*, а не на скриптовой логике.

Возможно, _hyperscript не поддерживает все стили и не обеспечивает все потребности скриптов, но он отлично подходит для приложений HDA. Пусть это небольшой и малоизвестный язык программирования, но вам определенно стоит познакомиться с ним поближе.

Использование готовых компонентов

На этом наше знакомство с тремя вариантами скриптовой инфраструктуры, то есть кода, который вы пишете для расширения возможностей своих гипермедиа-управляемых приложений, подходит к концу. Тем не менее существует еще одна важная область, которую необходимо учитывать при обсуждении скриптов на стороне клиента: готовые компоненты. Речь идет о библиотеках JavaScript, написанных другими специалистами и предоставляющих ту или иную функциональность, например отображение модальных диалоговых окон.

Компоненты весьма популярны в мире веб-разработки, а такие библиотеки, как DataTables[1], предоставляют широкие возможности взаимодействия с минимальным объемом кода JavaScript на стороне пользователя. К сожалению, если такие библиотеки недостаточно хорошо интегрированы в веб-сайт, приложение начинает напоминать лоскутное одеяло. Кроме того, некоторые библиотеки выходят за рамки простой манипуляции с DOM и требуют интеграции с конечной точкой сервера, почти всегда с API данных JSON. Это означает, что вы не сможете построить гипермедиа-управляемое приложение только из-за того, что конкретный виджет требует чего-то иного. Недопустимо!

[1] https://datatables.net/

ВЕБ-КОМПОНЕНТЫ И НЕСТАНДАРТНЫЕ ЭЛЕМЕНТЫ

Веб-компоненты (Web Components) – собирательное название для нескольких стандартов: Custom Elements (кастомные, или нестандартные, элементы) и Shadow DOM, а также <template> и <slot>.

Все эти стандарты вносят свой вклад в полезную функциональность. Элементы <template> удаляют свой контент из документа, одновременно проводя его парсинг как HTML (в отличие от комментариев) и делая его доступным для JavaScript.

Нестандартные элементы определяют поведение инициализации и завершения при добавлении и удалении элементов, что раньше приходилось делать вручную или с помощью наблюдателей `MutationObserver`. Shadow DOM позволяет инкапсулировать элементы, сохраняя чистоту «светлой» («нетеневой») DOM.

Тем не менее воспользоваться этими преимуществами часто оказывается непросто. Некоторые затруднения связаны с проблемами роста новых активно разрабатываемых стандартов (как проблемы доступности Shadow DOM). Другие возникают в результате того, что веб-компоненты пытаются одновременно взять на себя слишком много ролей.

- *Механизм расширения HTML*. Для этой цели каждый нестандартный элемент представляет собой тег, добавляемый в язык.
- *Механизм жизненного цикла для поведения*. Такие методы, как createdCallback, connectedCallback и т. д., позволяют добавлять поведение к элементам, при этом их не нужно вызывать вручную при добавлении этих элементов.
- *Единица инкапсуляции*. Shadow DOM инкапсулирует элементы от их окружения.

В результате, если вам нужно что-то одно из этого списка, все остальное идет за компанию. Если вы присоединяете какое-нибудь поведение к элементам с использованием обратных вызовов жизненного цикла, необходимо создать новый тег; это означает, что для элемента не удастся определить несколько вариантов поведения. Кроме того, добавляемые элементы изолируются от элементов, уже присутствующих на странице, что создаст проблемы, если они должны быть связаны отношениями ARIA.

Когда следует использовать веб-компоненты? Есть хорошее правило – спросите себя: «Может ли *это* быть встроенным элементом HTML»? Например, редактор кода может, потому что в HTML уже существуют элементы `<textarea>` и `contenteditable`. Кроме того, у полноценного редактора кода будет много дочерних элементов, которые все равно не будут информативны. Можно воспользоваться такими средствами, как

Shadow DOM[1], для инкапсуляции этих элементов[2]. Также можно создать нестандартный элемент[3] `<code-area>`, который можно расположить в любом месте страницы.

Возможности интеграции

Лучшие библиотеки JavaScript для построения гипермедиа-управляемых приложений — это такие библиотеки, которые:

- изменяют DOM, но не взаимодействуют с сервером в формате JSON;
- соблюдают нормы HTML (например, используют элементы `input` для хранения значений);
- инициируют множество нестандартных событий при обновлении модели библиотекой.

Последний пункт с выдачей многих нестандартных событий (вместо альтернативы в виде использования множества методов и обратных вызовов) особенно важен, так как эти нестандартные события могут передаваться или прослушиваться без дополнительного связующего кода, написанного на скриптовом языке.

Рассмотрим два разных подхода к написанию скриптов — с обратными вызовами JavaScript и с событиями.

Чтобы описание было более конкретным, реализуем улучшенное диалоговое окно подтверждения для кнопки Delete, созданной с Alpine в предыдущем разделе. В исходном примере использовалась функция `confirm()`, встроенная в JavaScript, которая выводит минималистичное системное диалоговое окно подтверждения. Заменим эту функцию популярной библиотекой JavaScript SweetAlert2, которая выводит более привлекательное окно подтверждения. В отличие от функции `confirm()`, которая блокирует выполнение и возвращает логический признак (`true`, если пользователь подтвердил операцию, `false` в противном случае), SweetAlert2 возвращает объект `Promise` — механизм JavaScript для подключения обратного вызова при завершении асинхронного действия (например, ожидания подтверждения или отказа пользователя от выполнения действия).

---

[1] https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_shadow_DOM

[2] Учтите, что Shadow DOM — относительно новая функциональность веб-платформы, которая все еще находится в разработке на момент написания книги. В частности, известно о ряде ошибок доступности, которые могут возникать при взаимодействии элементов внутри и вне теневого корневого элемента.

[3] https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_custom_elements

Интеграция с обратными вызовами

После установки SweetAlert2 как библиотеки открывается доступ к объекту `Swal`, который содержит функцию `fire()` для инициирования вывода оповещения. Методу `fire()` можно передать аргументы, которые определяют внешний вид кнопок в диалоговом окне подтверждения, текст в заголовке диалогового окна и т. д. Сейчас мы не будем отвлекаться на подробное описание, но вскоре вы увидите, как выглядит это диалоговое окно.

Таким образом, после установки библиотеки SweetAlert2 можно будет заменить ею вызов функции `confirm()`. После этого необходимо изменить структуру кода, чтобы *обратный вызов* передавался методу `then()` объекта `Promise`, возвращаемого `Swal.fire()`. Подробное рассмотрение `Promise` выходит за рамки этой главы — достаточно сказать, что этот обработчик события будет вызываться при подтверждении или отказе пользователя от выполнения действия. Если пользователь подтвердил действие, то свойство `result.isConfirmed` будет равно `true`.

С учетом всего сказанного обновленный код будет выглядеть примерно так:

Листинг 154. Диалоговое окно подтверждения с обратными вызовами

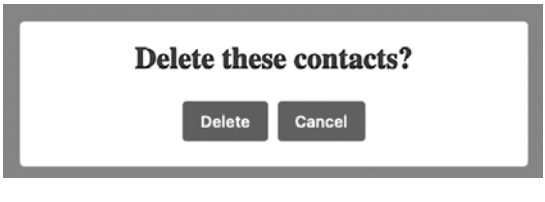
```
<button type="button" class="bad bg color border"
 @click="Swal.fire({ ❶
         title: 'Delete these contacts?', ❷
         showCancelButton: true,
         confirmButtonText: 'Delete'
        }).then((result) => { ❸
         if (result.isConfirmed) {
          htmx.ajax('DELETE', '/contacts', { source: $root, target:
document.body })
         }
       });"
>Delete</button>
```

❶ Вызывает функцию Swal.fire().

❷ Настраивает диалоговое окно (Delete these contacts? — «Удалить эти контакты?»).

❸ Обрабатывает результат выбора пользователя.

После этого по щелчку на кнопке в веб-приложении появляется новое диалоговое окно.



Намного приятнее, чем системное диалоговое окно подтверждения. И все же что-то здесь не так. Слишком много кода приходится писать просто ради того, чтобы вызвать слегка улучшенную версию `confirm()`, вы так не думаете? И JavaScript-код `htmx`, который мы используем здесь, все еще выглядит грубо. Естественнее переместить `htmx` в атрибуты `button`, как раньше, а затем инициировать запрос через события.

Выберем другой путь и посмотрим, что получится.

Интеграция с использованием событий

Чтобы подчистить код, выделим код `Swal.fire()` в отдельную функцию JavaScript, которая будет называться `sweetConfirm()`. `sweetConfirm()` получает параметры, передаваемые методу `fire()`, а также элемент, подтверждающий действие. Главное отличие заключается в том, что новая функция `sweetConfirm()` не обращается с вызовами `htmx` напрямую, а инициирует событие `confirmed` для кнопки, когда пользователь подтверждает удаление.

Новая функция JavaScript выглядит так:

Листинг 155. Диалоговое окно подтверждения на основе событий

```
function sweetConfirm(elt, config) {
   Swal.fire(config) ❶
     .then((result) => {
         if (result.isConfirmed) {
           elt.dispatchEvent(new Event('confirmed')); ❷
         }
     });
}
```

❶ Конфигурация передается функции fire().

❷ Если пользователь подтвердил действие, инициируется событие confirmed.

При наличии такого метода код кнопки удаления становится более компактным. Мы можем удалить весь код SweetAlert2, который содержался в атрибуте Alpine `@click`, и просто вызвать новый метод `sweetConfirm()`, передавая в аргументах `$el`, что в синтаксисе Alpine обозначает текущий элемент, для которого выполняется скрипт, и точное описание конфигурации диалогового окна.

Если пользователь подтвердит действие, для кнопки будет инициировано событие `confirmed`. А это значит, что можно вернуться к знакомым атрибутам `htmx`! То есть переместить запрос `DELETE` в атрибут `hx-delete` и использовать `hx-target` для выбора `body` в качестве цели. Затем (и это самый важный шаг) событие `confirmed`, инициируемое в функции `sweetConfirm()`, можно использовать для инициирования запроса, но с добавлением для него атрибута `hx-trigger`.

Код выглядит так:

Листинг 156. Диалоговое окно подтверждения на основе событий

```
<button type="button" class="bad bg color border"
    hx-delete="/contacts" hx-target="body" hx-trigger="confirmed" ❶
    @click="sweetConfirm($el, ❷
        { title: 'Delete these contacts?', ❸
         showCancelButton: true,
         confirmButtonText: 'Delete'})">
```

❶ Атрибуты htmx вернулись на свое место.
❷ Кнопка передается функции, чтобы событие могло быть инициировано для нее.
❸ Передается информация конфигурации SweetAlert2.

Как видите, код на основе событий получается более чистым и, несомненно, лучше соответствует духу HTML. Условием этой более чистой реализации становится то, что новая функция `sweetConfirm()` инициирует событие, которое может прослушиваться `htmx`.

Вот почему расширенная модель событий так важна при выборе библиотеки для работы — как с `htmx`, так и с приложениями HDA в целом.

К сожалению, из-за преобладающей тенденции ставить на первое место JavaScript многие библиотеки действуют по образцу SweetAlert2: они ожидают передачи обратного вызова. В таких случаях можно использовать прием, описанный выше: упаковать библиотеку в функцию, которая инициирует события в обратном вызове, чтобы библиотека стала более гипермедиа- и `htmx`-ориентированной.

Прагматичные скрипты

> “ При возникновении конфликтов на первое место ставятся интересы пользователей; далее в порядке убывания приоритета — авторы, создатели реализации, разработчики спецификаций и теоретическая чистота.
>
> *W3C HTML Design Principles § 3.2 Priority of Constituencies*

Мы рассмотрели некоторые инструменты и приемы для написания скриптов в гипермедиа-управляемых приложениях. Как же выбрать из них наиболее подходящий? Печально, но на этот вопрос никогда не будет единственно верного ответа.

Вы привыкли к решениям, ограничивающимся «ванильным» JavaScript, — возможно, из-за политики компании? Тогда для скриптов в своих приложениях HDA можете использовать «ванильную» версию.

У вас больше свободы действий и вам нравится оформление Alpine.js? Это отличный способ добавить более структурированный, локализованный код JavaScript в свое приложение, который к тому же предоставляет удобную реактивную функциональность.

Вы готовы рискнуть в выборе технологии? Возможно, стоит присмотреться к _hyperscript (мы определенно так считаем).

Иногда даже можно рассмотреть возможность применения двух (и более) этих решений. У каждого варианта есть свои достоинства и недостатки, все они относительно компактны и автономны, так что выбор разных инструментов под конкретные задачи может оказаться лучшим решением.

В общем случае мы рекомендуем *прагматичный* подход к написанию скриптов: то, что кажется вам верным, скорее всего, верно (или по крайней мере *достаточно* верно) для вас. Вместо выбора определенного подхода к написанию скриптов уделите внимание более общим вопросам.

- Избегайте взаимодействия с сервером через JSON Data API.
- Избегайте хранения больших объемов состояния за пределами DOM.
- Отдавайте предпочтение событиям вместо жестко запрограммированных обратных вызовов или вызовов методов.

И даже здесь веб-разработчику иногда приходится подчиняться обстоятельствам. Идеальный виджет для вашего приложения существует, но использует JSON Data API? Ничего страшного. Просто не увлекайтесь подобной практикой.

Заметки об HTML: HTML подходит для приложений

Среди разработчиков бытует мнение, что язык HTML проектировался для «документов», а для «приложений» он не подходит. В действительности гипермедиа — современная, проработанная архитектура для приложений; к тому же она позволит раз и навсегда разобраться с искусственным делением «приложение/документ».

> “ Говоря о «гипертексте», я имею в виду одновременное представление информации и элементов управления, посредством которого пользователь получает возможность выбора действий.
>
> *Рой Филдинг, «A little REST and Relaxation»*[1]

[1] https://www.slideshare.net/royfielding/a-little-rest-and-relaxation

HTML позволяет включать в документы богатую мультимедийную информацию, в том числе графику, аудио, видео, программы JavaScript, векторную графику и (с некоторой поддержкой) 3D-объекты. Но что еще важнее, это позволяет встраивать в такие документы интерактивные элементы управления, чтобы сама информация стала приложением, через которое пользователь работает с ней.

Только подумайте: разве не удивительно, что в одном приложении, которое работает на всех видах компьютеров и ОС, можно читать новости, создавать видеочаты, формировать документы, входить в виртуальную реальность и выполнять практически любые повседневные вычислительные задачи?

К сожалению, именно интерактивные возможности HTML — наименее проработанная сторона этого языка. По какой-то причине, хотя HTML получил развитие до версии 5 и стал HTML Living Standard (то есть текущим стандартом), обзаведясь многими революционными возможностями, взаимодействия с данными в нем в основном ограничиваются ссылками и формами. Разработчикам приходится самим заниматься расширением HTML, и они хотят делать это так, чтобы не имитировать классические «нативные» инструменты.

> “
> - В ПО не должны были использоваться нативные инструментарии.
> - Многолетняя разработка UI-библиотек для Windows так и не привела к реальному их применению на уровнях ниже веб-среды.
> - Хотели окно только ради забавы? У нас был для этого инструмент: он назывался ELECTRON.
> - «Да, мне очень хотелось бы написать четыре РАЗНЫЕ копии одного пользовательского интерфейса» — заявления из арсенала Совершенно Невменяемых.
>
> *Леа Кларк (Leah Clark) @leah@tilde.zone*

API ДАННЫХ JSON И ГИПЕРМЕДИА-УПРАВЛЯЕМЫЕ ПРИЛОЖЕНИЯ

До сих пор мы ограничивались применением гипермедиа для построения гипермедиа-управляемых приложений (HDA). При их создании мы оставались в рамках нативной сетевой веб-архитектуры, использовали ее возможности и строили RESTful-систему в исходном смысле этого термина.

Однако приходится признать, что сегодня при построении многих веб-приложений этот подход не используется. Вместо этого применяются фронтенд-библиотеки SPA, такие как React, и приложения взаимодействуют с сервером через JSON API. JSON API почти никогда не использует концепции гипермедиа. JSON API обычно представляют собой *API данных*, то есть API, которые просто возвращают клиенту структурированные данные предметной области без сопутствующей информации элементов управления гипермедиа. Сам клиент должен знать, как интерпретировать данные JSON: какие конечные точки связываются с данными, как должны интерпретироваться определенные поля и т. д.

На самом же деле мы *создавали* API для Contact.app.

Но как такое может быть? Мы всего лишь создавали веб-приложение с обработчиками, которые просто возвращают HTML.

Почему это API?

Оказывается, Contact.app действительно предоставляет API. Просто это API *гипермедиа*, понятный для *клиента гипермедиа* (то есть браузера). Мы строим API для браузера, чтобы взаимодействовать с ним через протокол HTTP, и благодаря чудесным свойствам HTML и гипермедиа браузеру не нужно ничего знать об API *гипермедиа,* кроме URL точки входа: все действия и выводимая автономная информация содержатся в ответах HTML.

Создавать RESTful-приложения для веб по такой схеме настолько естественно и просто, что можно вообще не рассматривать его как API, но мы уверяем, что это именно он.

API гипермедиа и API данных JSON

Итак, у нас есть API гипермедиа для Contact.app. Можно ли также предоставить API данных для Contact.app?

Конечно! Существование API гипермедиа *никак не означает*, что нельзя *также* иметь API данных. Собственно, это обычная ситуация в традиционных веб-приложениях: существует «веб-приложение», вход в которое осуществляется через URL точки входа, например https://mywebapp.example.com/. Также имеется отдельный JSON API, доступный по другому URL, например https://api.mywebapp.example.com/v1.

Это абсолютно логичный способ разделения гипермедиа-интерфейса вашего приложения и API данных, которые вы предоставляете другим, не гипермедиа-клиентам.

Зачем добавлять API данных к API гипермедиа? Потому что *не гипермедиа-клиентам* также может потребоваться взаимодействовать с приложением.

Например:

- Представим, что у вас имеется мобильное приложение, которое было построено без использования Hyperview. Это приложение должно как-то взаимодействовать с сервером, а существующий HTML API почти наверняка будет не лучшим вариантом! Вам нужен программный доступ к системе через API данных, и JSON — логичный выбор для этого.
- Возможно, у вас есть автоматизированный скрипт, который должен регулярно взаимодействовать с системой. Например, задание массового импорта данных, которое выполняется каждую ночь и импортирует/синхронизирует тысячи контактов. Хотя написать скриптовое решение для HTML API можно, делать этого не стоит: парсинг HTML в скриптах ненадежен и утомителен. Для такого сценария использования лучше иметь простой JSON API.
- Возможно, существуют сторонние клиенты, которым ненобходимо каким-то образом интегрировать данные вашей системы. Возможно, партнер должен каждую ночь синхронизировать данные. Как и в примере с массовым импортом, это не лучший сценарий использования для API на основе HTML, и разумнее предоставить что-то более подходящее для скриптовой реализации.

В таких ситуациях применение API данных JSON оправданно: в любом случае API не потребляется клиентом гипермедиа, так что API гипермедиа на основе HTML будет слишком неэффективным и сложным для клиента. Простой API данных JSON вполне подходит, и как обычно, мы рекомендуем выбирать инструмент, подходящий для текущей задачи.

«ЧТО?! ВЫ ХОТИТЕ, ЧТОБЫ Я ПАРСИЛ HTML?!»

Когда мы продвигаем гипермедиа-подход к созданию веб-приложений в онлайн-сообществе, часто возникает путаница: наши собеседники думают, что им придется парсить ответы HTML от сервера, а затем сбрасывать данные в фреймворк SPA или мобильные приложения.

Конечно, это глупо.

Речь о другом: API гипермедиа следует использовать с *клиентом гипермедиа* (например, браузером), который будет интерпретировать гипермедиа-ответ и представлять его пользователю. API гипермедиа нельзя просто «налепить» поверх существующего решения SPA. Для его эффективного внедрения потребуется сложный клиент гипермедиа, такой как браузер.

Если вы пишете код для парсинга гипермедиа только для того, чтобы затем интерпретировать гипермедиа как данные и передать модели на стороне клиента, скорее всего, вы делаете что-то не так.

Чем API гипермедиа отличаются от API данных

Ненадолго представим, что в нашем приложении наряду с API гипермедиа *будет* реализован API данных. На этой стадии у разработчиков может возникнуть вопрос: зачем поддерживать оба API? Почему не ограничиться единственным API — API данных JSON, с которым будут взаимодействовать все клиенты?

Разве наличие обоих типов API не избыточно для приложения?

Разумное замечание: мы рекомендуем в случае необходимости поддерживать в веб-приложении несколько API, и это действительно может привести к некоторой избыточности в коде. Тем не менее у каждого вида API существуют свои преимущества, и более того, для них устанавливаются разные требования.

Поддерживая оба типа API по отдельности, можно сохранить сильные стороны обоих типов, одновременно обеспечивая четкое разделение разных стилей программирования и инфраструктурных потребностей.

Сравним требования к JSON API с требованиями к гипермедийному API.

Требования к JSON API	Требования к API гипермедиа
Должен оставаться стабильным по времени: вы не сможете изменять API по своему усмотрению, иначе рискуете нарушить работоспособность клиентов, использующих API и ожидающих определенного поведения от некоторых конечных точек	Не обязан оставаться стабильным по времени: все URL определяются по ответам HTML, так что можно намного активнее изменять форму API

Требования к JSON API	Требования к API гипермедиа
Требует контроля версии: при внесении серьезных изменений необходимо управлять версией API, чтобы клиенты, использующие старую версию API, продолжали работать	Отсутствие проблем с контролем версии: еще одна сильная сторона гипермедиа
Требует ограничения по частоте обращений: так как API данных часто используются другими клиентами, а не только внутренним веб-приложением, запросы должны быть ограничены по частоте (по каждому пользователю), чтобы один клиент не перегружал систему	Скорее всего, ограничения частоты не будут актуальными, кроме предотвращения распределенных атак отказа в обслуживании (DDoS)
API должен иметь общую природу: так как он предназначен для *всех* клиентов, а не только для вашего веб-приложения, следует избегать специализированных конечных точек, которые определяются потребностями приложения. API должен быть общим и достаточно выразительным, чтобы удовлетворить как можно больше потенциальных потребностей клиента	API может быть тесно связан с потребностями приложения: так как он спроектирован только для конкретного веб-приложения, а API обнаруживается средствами гипермедиа, можно добавлять и удалять специализированные конечные точки для конкретных функций или требований к оптимизации в приложении
Такие API обычно используют аутентификацию на основе маркеров (более подробно мы рассмотрим ее позже)	Для управления аутентификацией обычно используется сеансовый cookie, который создается на странице входа

Две разновидности API имеют разные достоинства и недостатки, так что есть смысл применять и то и другое. Подход гипермедиа можно использовать, чтобы «подстроить» API под «форму» приложения, а API данных — для других, негипермедийных клиентов: мобильных устройств, партнеров по интеграции и т. д.

Обратите внимание: разделяя эти два API, вы избавляетесь от необходимости постоянно изменять общий API данных под конкретные потребности приложения. API данных может поддерживать стабильность и надежность, чтобы не требовать новой версии с каждой добавляемой функцией.

И в этом заключается главное преимущество отделения API данных от API гипермедиа.

API ДАННЫХ JSON И JSON REST API

К сожалению, так сложилось, что то, что мы называем API данных JSON, часто называют «REST API». Это парадоксально, потому что при любом сколько-нибудь внимательном прочтении работы Роя Филдинга, определяющей смысл термина «REST», будет понятно, что подавляющее большинство JSON API не соответствуют принципам REST. Даже близко.

> “Меня раздражает, что очень многие люди называют любой интерфейс на основе HTTP «REST API». Самый свежий пример — SocialSite REST API. Это RPC. Все в нем указывает на RPC. Здесь столько случайных связей, что ему следовало бы присвоить рейтинг «только для взрослых».
>
> Что необходимо сделать, чтобы архитектурный стиль REST ясно выражал, что гипертекст является ограничением? Иначе говоря, если ядро состояния приложения (а следовательно, API) не управляется гипертекстом, то API не может соответствовать принципам REST и не может быть RESTful. Точка. Возможно, в какой-то мануал закралась ошибка, которую нужно исправить?
>
> *Рой Филдинг.* https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

История о том, как определение «REST API» в отрасли стало восприниматься как «JSON API», долгая и неприглядная и выходит за рамки книги. Тем не менее, если вам она интересна, вы можете ознакомиться с очерком одного из авторов статьи *How Did REST Come To Mean The Opposite of REST?* на сайте `htmx`[1].

В этой книге мы будем использовать термин «API данных» для описания таких JSON API, при этом понимая, что многие в отрасли в обозримом будущем продолжат называть ее «REST API».

Добавление API данных JSON в Contact.app

Хорошо, как же добавить API данных JSON в наше приложение? В одной из техник, популяризированных веб-фреймворком Ruby on Rails, используются те же конечные точки URL, что и в приложении гипермедиа, но для определения того, какое представление требуется клиенту — JSON или HTML, применяют

[1] https://htmx.org/essays/how-did-rest-come-to-mean-the-opposite-of-rest/

заголовок HTTP `Accept`. Заголовок HTTP `Accept` позволяет клиенту указать, какие типы MIME (Multipurpose Internet Mail Extensions), то есть какие типы файлов, он хочет получить от сервера: JSON, HTML, текст и т. д.

Таким образом, если клиент хочет получить все контакты в представлении JSON, он может выдать запрос `GET`, который выглядит примерно так:

Листинг 157. Запрос всех контактов в представлении JSON

```
Accept: application/json

GET /contacts
```

Если принять этот паттерн, то обработчик запроса для `/contacts/` необходимо обновить, чтобы он проверял заголовок и в зависимости от значения возвращалось нужное представление — JSON или HTML. В Ruby on Rails поддержка этого паттерна встроена во фреймворк, что позволяет легко переключаться на нужный тип MIME.

К сожалению, наш опыт использования этого паттерна оставляет желать лучшего по причинам, которые должны быть понятны с учетом описанных выше различий между API данных и API гипермедиа: у них разные требования, часто они принимают разные «формы», и попытка втиснуть их в один набор URL создает лишнюю напряженность в коде приложения.

С учетом разных требований двух API и нашего опыта управления разными API в подобных ситуациях мы считаем, что разделение API — и, как следствие, выделение для API данных JSON отдельного набора URL — будет правильным решением. Это позволит двум API эволюционировать независимо, и можно будет совершенствовать каждый из них по отдельности с учетом их сильных сторон.

Выбор корневого URL для API

Так как мы собираемся отделить маршруты API данных JSON от обычных маршрутов гипермедиа, где их размещать? Важно позаботиться о том, чтобы API четко версионировался независимо от выбранной схемы.

Если присмотреться к окружению, во многих местах для API используется субдомен (вида https://api.mywebapp.example.com), а в субдомене часто кодируется информация о версии:

https://v1.api.mywebapp.example.com.

Хотя такое решение выглядит уместно в больших инструментах, для нашего маленького приложения Contact.app оно кажется избыточным. Вместо субдоме-

нов, что создает проблемы для локальной разработки, мы будем использовать подпути в пределах существующего приложения:

- `/api` в качестве корня функциональности API данных;
- `/api/v1` как точку входа для версии 1 API данных.

Если (и когда) мы решим изменить версию API, можно будет перейти на `/api/v2` и т. д.

Конечно, этот способ неидеален, но он подойдет для нашего простого приложения и его можно адаптировать к субдоменному решению или другим методам, когда наше приложение Contact.app захватит интернет и мы сможем позволить себе большую команду разработчиков API :).

Первая конечная точка JSON: вывод всех контактов

Добавим первую конечную точку API данных. Она будет обрабатывать запрос HTTP `GET` к `/api/v1/contacts` и возвращать список JSON всех контактов в системе. Код новой точки отчасти напоминает исходный код маршрута гипермедиа `/contacts`: мы загружаем все контакты из базы данных контактов, а затем рендерим текст как ответ.

Мы также воспользуемся одной удобной возможностью Flask: если вы просто возвращаете объект из обработчика, то он сериализуется (то есть преобразуется) в ответ JSON. Это позволяет очень легко строить простые JSON API при помощи Flask!

Листинг 158. API данных JSON для получения всех контактов

```
@app.route("/api/v1/contacts", methods=["GET"]) ❶
def json_contacts():
  contacts_set = Contact.all()
  contacts_dicts = [c.__dict__ for c in contacts_set] ❷
  return {"contacts": contacts_dicts} ❸
```

❶ API данных JSON получает собственный путь, начинающийся с /api.
❷ Массив контактов преобразуется в массив простых объектов словарей.
❸ Возвращает словарь, содержащий свойство contacts всех контактов.

Этот код Python может показаться незнакомым, если у вас нет опыта разработки на Python, но здесь мы просто преобразуем данные контактов в массив простых пар «имя/значение» и возвращаем полученный массив во внешнем объекте как свойство `contacts`. Flask автоматически сериализует этот объект в формат JSON.

Когда все это будет сделано, при выдаче запроса HTTP `GET` к `/api/v1/contacts` мы получим ответ, который выглядит примерно так:

Листинг 159. Примеры данных из нашего API

```
{
 "contacts": [
  {
   "email": "carson@example.com",
   "errors": {},
   "first": "Carson",
   "id": 2,
   "last": "Gross",
   "phone": "123-456-7890"
  },
  {
   "email": "joe@example2.com",
   "errors": {},
   "first": "",
   "id": 3,
   "last": "",
   "phone": ""
  },
  ...
 ]
}
```

Итак, как видите, у нас появился способ получения относительно простого представления контактов в формате JSON через запрос HTTP. Не идеально, но для начала неплохо. Разумеется, этого достаточно для написания базовых автоматизированных скриптов. Например, при помощи этого API данных можно:

- каждую ночь передавать контакты в другую систему;
- создавать резервную копию контактов в локальном файле;
- автоматизировать рассылку электронной почты среди контактов.

Существование такого небольшого API данных JSON открывает значительные возможности для автоматизации, которые было бы труднее обеспечить с существующим API гипермедиа.

Добавление контактов

Перейдем к следующей функциональности: добавлению новых контактов. И снова наш код будет в чем-то похож на код, написанный для обычного веб-приложения. Однако вы увидите, что JSON API и API гипермедиа приложения в этом отношении тоже ощутимо расходятся.

В веб-приложении потребуется отдельный путь `contacts/new` для формы HTML, предназначенной для создания нового контакта. В веб-приложении мы решили выдать запрос `POST` к тому же пути просто ради единообразия.

В случае JSON API такой путь не нужен: JSON API «просто существует» — ему не требуется никакое представление гипермедиа для создания нового контакта. Вы просто знаете, куда следует отправить запрос `POST` для создания контакта (вероятнее всего, из документации для API), и это все.

Благодаря этому можно разместить обработчик создания по тому же пути, что и обработчик вывода: `/api/v1/contacts`, но чтобы он реагировал только на запросы HTTP `POST`.

Приведенный код относительно прост: заполняем новый контакт информацией из запроса `POST`, пытаемся сохранить его и — если попытка не будет успешной — выводим сообщение об ошибках.

Код выглядит так:

Листинг 160. Добавление контактов через JSON API

```
@app.route("/api/v1/contacts", methods=["POST"]) ❶
def json_contacts_new():
  c = Contact(None, request.form.get('first_name'),
request.form.get('last_name'), request.form.get('phone'),
        request.form.get('email')) ❷
  if c.save(): ❸
    return c.__dict__
  else:
    return {"errors": c.errors}, 400 ❹
```

❶ Обработчик доступен по тому же пути, что и первый для JSON API, но обрабатывает запросы POST.

❷ Создает новый экземпляр Contact на основании значений, переданных с запросом.

❸ Пытается сохранить контакт, и в случае успеха рендерит его в виде объекта JSON.

❹ Если при сохранении произошла ошибка, рендерится объект с ошибками с кодом ответа 400 (неправильный запрос).

В каком-то отношении код похож на обработчик `contacts_new()` из веб-приложения; мы создаем контакт и пытаемся сохранить его. В других отношениях он сильно отличается.

- При успешном создании перенаправления не происходит, потому что не используется клиент гипермедиа (например, браузер).
- В случае неправильного запроса просто возвращается код ответа для ошибки `400 Bad Request` (неправильный запрос). Этот результат сильно отличается от веб-приложения, где форма рендерится заново с сообщением об ошибках.

Такие различия со временем накапливаются, и идея поддержания JSON API и API гипермедиа с одним набором URL выглядит все менее и менее привлекательно.

Просмотр подробной информации о контактах

Теперь сделаем так, чтобы клиент JSON API загружал подробную информацию одного контакта. Для этой функциональности естественным образом используется запрос HTTP `GET`; мы последуем схеме, выбранной для предыдущего веб-приложения, и используем путь `/api/v1/contacts/<contact id>`. Таким образом, например, чтобы просмотреть подробную информацию о контакте с идентификатором 42, следует выдать запрос HTTP `GET` к `/api/v1/contacts/42`.

Код выглядит достаточно просто.

Листинг 161. Получение подробной информации контакта в формате JSON

```
@app.route("/api/v1/contacts/<contact_id>", methods=["GET"]) ❶
def json_contacts_view(contact_id=0):
  contact = Contact.find(contact_id) ❷
  return contact.__dict__ ❸
```

❶ Добавляет новый маршрут GET с путем, который должен использоваться для просмотра подробной информации контакта.

❷ Ищет контакт по идентификатору, переданному в пути.

❸ Преобразует контакт в словарь, чтобы его можно было отрендерить как ответ JSON.

Ничего сложного; контакт обнаруживается по идентификатору, переданному в пути контроллеру. Затем информация контакта рендерится в формате JSON. Оцените простоту этого кода!

А теперь добавим функции обновления и удаления контактов.

Обновление и удаление контактов

Как и в случае с конечной точкой API для создания контактов, поскольку HTML UI для них не существует, можно воспользоваться путем `/api/v1/contacts/<contact id>`. Мы применим метод HTTP `PUT` для обновления контакта и метод `DELETE` для удаления контакта.

Код обновления будет практически идентичен обработчику создания, за исключением того, что вместо создания нового контакта мы будем находить контакт по идентификатору и обновлять его поля. В этом смысле мы просто объединяем код обработчика создания с кодом обработчика просмотра.

Листинг 162. Обновление контакта в JSON API

```
@app.route("/api/v1/contacts/<contact_id>", methods=["PUT"]) ❶
def json_contacts_edit(contact_id):
  c = Contact.find(contact_id) ❷
  c.update(request.form['first_name'], request.form['last_name'],
```

```
request.form['phone'], request.form['email']) ❸
  if c.save(): ❹
    return c.__dict__
  else:
    return {"errors": c.errors}, 400
```

❶ Обрабатывает запросы PUT к URL для заданного контакта.
❷ Ищет контакт по идентификатору, переданному в пути.
❸ Данные контакта обновляются значениями, включенными в запрос.
❹ Дальнейшая логика идентична логике обработчика json_contacts_create().

И снова благодаря встроенной функциональности Flask реализация получается простой.

Перейдем к удалению контакта. Эта реализация получается еще более простой: как и в случае с обработчиком обновления, она ищет контакт по идентификатору, а затем удаляет его. В этой точке можно вернуть простой объект JSON как признак успеха.

Листинг 163. Удаление контакта в JSON API

```
@app.route("/api/v1/contacts/<contact_id>", methods=["DELETE"]) ❶
def json_contacts_delete(contact_id=0):
  contact = Contact.find(contact_id)
  contact.delete() ❷
  return jsonify({"success": True}) ❸
```

❶ Обрабатывает запросы DELETE к URL для заданного контакта.
❷ Ищет контакт и вызывает для него метод delete().
❸ Возвращает простой объект JSON как признак успешного удаления контакта.

После этого у нас появляется простой маленький API данных JSON, который существует наряду с обычным веб-приложением, но при этом он как следует отделен от главного приложения, что позволяет ему эволюционировать независимо по мере надобности.

Другие соображения, касающиеся API данных

Если бы мы создавали реальный рабочий JSON API, нам пришлось бы сделать еще много всего. Как минимум реализовать следующие функции:

- ограничение частоты запросов, которое важно для любого общедоступного API данных, чтобы предотвращать злоупотребления со стороны клиента;
- механизм аутентификации. (В веб-приложении его тоже не было!);

- поддержка разбивки на страницы для данных контактов;
- некоторые второстепенные функции, такие как рендеринг корректного ответа `404 Not Found` (Не найдено), если кто-то выдаст запрос с несуществующим идентификатором контакта.

Эти вопросы выходят за рамки книги, но мы затронем один из них — аутентификацию, — чтобы продемонстрировать различия между API гипермедиа и API JSON. Чтобы приложение было безопасным, необходимо добавить *аутентификацию* — механизм определения, от кого поступил запрос, и *авторизацию* — механизм определения, есть ли у этого субъекта право отправлять запрос.

Пока отложим вопрос авторизации и ограничимся аутентификацией.

Аутентификация в веб-приложениях

В веб-приложениях HTML для выполнения аутентификации традиционно используется страница входа, на которой предлагается ввести имя пользователя (часто используется адрес электронной почты) и пароль. Этот пароль проверяется по основе данных (хешированных) паролей, чтобы установить, является ли пользователь тем, за кого он себя выдает. Если пароль верный, создается *сеансовый cookie*, идентифицирующий пользователя. Этот объект cookie отправляется с каждым запросом, который выдается пользователем к веб-приложению, чтобы приложение знало, от кого поступил запрос.

ОБЪЕКТЫ COOKIE В HTTP

Объекты cookie относятся к числу необычных возможностей HTTP. В каком-то отношении они нарушают цель отсутствия состояния – основной компонент RESTful-архитектуры: сервер часто использует сеансовые cookie в качестве индекса для состояния, хранимого сервером «на стороне», например в кэше последних действий, выполняемых пользователем.

Тем не менее cookie оказались исключительно полезными, и разработчики обычно не слишком жалуются на них (а какие еще есть варианты?). Это интересный пример (относительно) здорового прагматизма в веб-разработке.

В отличие от стандартного подхода к аутентификации в веб-приложениях, JSON API обычно использует аутентификацию *на основе маркеров*: маркер аутентификации создается таким механизмом, как OAuth, и передается (обычно в заголовке HTTP) с каждым запросом, выдаваемым клиентом.

На высоком уровне эта схема напоминает то, что происходит при обычной аутентификации веб-приложений: маркер создается, а затем становится частью каждого запроса. Однако на практике эти механизмы радикально отличаются.

- Объекты cookies являются частью спецификации HTTP и могут легко *назначаться* сервером HTTP.
- Маркеры аутентификации JSON требуют *сложных механизмов передачи*, таких как OAuth.

Различия в механиках для установления аутентификации — еще одна убедительная причина для разделения JSON API и API гипермедиа.

«Очертания» двух API

При построении собственных API мы заметили, что во многих случаях JSON API требует меньшего количества конечных точек, чем API гипермедиа: например, нам не нужен обработчик `/contacts/new`, предоставляющий представление гипермедиа для создания контактов.

Необходимо учесть и такое свойство API гипермедиа, как достигнутое повышение эффективности: мы выделили подсчет общего количества контактов в отдельную конечную точку и реализовали паттерн «Отложенная загрузка» для улучшения субъективной производительности приложения. Если API гипермедиа и JSON API совместно используют одинаковые пути, то следует ли публиковать этот API также как конечную точку JSON?

Может, да... а может, и нет. Это была конкретная потребность нашего приложения, и при отсутствии запроса от пользователя JSON API нет смысла включать ее для потребителей JSON. А что, если проблемы с производительностью `Contact.count()`, которые решались при помощи паттерна «Отложенная загрузка», вдруг исчезнут? В гипермедиа-управляемом приложении мы бы просто вернулись к старому коду и включили подсчет прямо в запрос к `/contacts`. Тогда конечную точку `contacts/count` и всю связанную с ней логику можно будет удалить. Благодаря унифицированному интерфейсу гипермедиа система продолжит нормально работать.

А если бы JSON API был связан с API гипермедиа, а путь `/contacts/count` был опубликован как поддерживаемая конечная точка для JSON API? В таком случае просто удалить конечную точку не удастся: от нее может зависеть (негипермедийный) клиент. Здесь снова проявляется гибкость подхода гипермедиа, и мы видим, почему отделение JSON API от API гипермедиа позволяет использовать все преимущества этой гибкости.

Парадигма MVC (Model-View-Controller)

Об обработчиках нашего JSON API можно сказать одно: они относительно просты и стандартны. Большая часть черной работы по обновлению данных и т. д. выполняется в самой модели контакта: обработчики служат простыми соединительными звеньями, обеспечивающими взаимодействие между запросами HTTP и моделью.

Это идеальный контроллер в парадигме «модель — представление — контроллер», или MVC (Model-View-Controller), столь популярной на заре становления веб-среды: контроллер должен быть «тонким», а модель должна содержать большую часть логики в системе.

ПАТТЕРН «МОДЕЛЬ – ПРЕДСТАВЛЕНИЕ – КОНТРОЛЛЕР»

Паттерн проектирования «модель – представление – контроллер» относится к числу классических архитектурных паттернов в разработке ПО; он оказал огромное влияние на раннюю веб-разработку. Сейчас ему уже не придается такого значения, так как веб-разработка разделилась на фронтенд и бэкенд, но многие веб-разработчики еще хорошо знакомы с этим паттерном.

Традиционно паттерн MVC в веб-разработке выглядел так:

- **Модель.** Набор классов предметной области, реализующих всю логику и правила конкретной предметной области, для которой проектировалось приложение. Модель обычно поставляет «ресурсы», которые затем выводятся клиентам в виде HTML.
- **Представление.** Обычно реализовано в виде системы шаблонов на стороне клиента. Рендерит упомянутый выше HTML вывода для заданного экземпляра модели.
- **Контроллер.** Его задача – принимать запросы HTTP, преобразовывать их в содержательные запросы к модели и передавать соответствующим объектам модели. Затем HTML вывода возвращается клиенту в виде ответа HTTP.

Тонкие контроллеры упрощают разделение JSON API и API гипермедиа, поскольку вся важная логика находится в модели предметной области, общей для обоих. Это позволяет развивать два вида API по отдельности, сохраняя синхронизацию логики между ними.

С правильно построенными «тонкими» контроллерами и «толстыми» моделями обеспечивать синхронизацию двух API с возможностью их раздельной эволюции не так сложно и страшно, как может показаться на первый взгляд.

Заметки об HTML: микроформаты

Микроформаты[1] — стандарт для внедрения в HTML структурированных машиночитаемых данных. В нем используются классы для пометки определенных элементов как содержащих извлекаемую информацию, с соглашениями для извлечения типичных свойств (имя, URL, фото и т. д.) без участия классов. Добавляя эти классы в презентационную разметку HTML объекта, мы открываем возможность получения свойств из HTML. Например, следующий простой фрагмент HTML:

```
<a class="h-card" href="https://john.example">
 <img src="john.jpg" alt=""> John Doe
</a>
```

разбирается в JSON-подобную структуру парсером микроформатов:

```
{
 "type": ["h-card"],
 "properties": {
  "name": ["John Doe"],
  "photo": ["john.jpg"],
  "url": ["https://john.example"]
 }
}
```

Используя разные свойства и вложенные объекты, можно разметить каждый бит информации (например, о контакте) так, чтобы он стал машиночитаемым.

Как объяснялось в предыдущей главе, использование одного механизма для взаимодействия с человеком и машиной — не лучшая идея. Интерфейсы, обращенные к человеку и машине, могут ограничивать друг друга. Если вы хотите предоставить пользователям и разработчикам данные и действия, относящиеся к предметной области, JSON API станет отличным способом это сделать.

С микроформатами проблем будет намного меньше. Протокол или стандарт, требующий от веб-сайта реализации JSON API, создает высокий технический барьер. С другой стороны, любой веб-сайт можно дополнить вставками микроформатов простым добавлением нескольких классов. Другие встраиваемые в HTML форматы данных — микроданные, Open Graph — так же легко интегрируются в приложения. Это делает микроформаты удобными для межсайтовых систем (осмелимся сказать — систем в *масштабе веб-среды*), таких как IndieWeb[2], в которых они широко применяются.

---

[1] https://microformats.org/

[2] https://indieweb.org/

ЧАСТЬ III

Гипермедиа для мобильных устройств

HYPERVIEW: МОБИЛЬНАЯ ГИПЕРМЕДИА-ПЛАТФОРМА

Если кто-то из читателей считает, что архитектура гипермедиа является синонимом веба, веб-браузеров и HTML, мы простим это заблуждение. Несомненно, веб — самая большая система гипермедиа, а веб-браузер — самый популярный клиент гипермедиа. Из-за доминирующего положения веб-среды в обсуждениях гипермедиа легко забыть о том, что гипермедиа — общая концепция, которая может работать на всех типах платформ и приложений. В этой главе вы увидите, как архитектура гипермедиа применяется не в веб-среде, а в нативных мобильных приложениях.

Мобильные устройства как платформа устанавливают иные ограничения, нежели веб-среда. Они требуют других компромиссов и проектировочных решений. Тем не менее концепции гипермедиа, HATEOAS и REST можно напрямую применять для построения удобных мобильных приложений.

В этой главе мы поговорим о текущем состоянии мобильной разработки и о том, как архитектура гипермедиа может решить имеющиеся в ней проблемы. Затем рассмотрим работу гипермедиа на мобильных устройствах с помощью Hyperview, фреймворка для создания мобильных приложений, использующего архитектуру гипермедиа. Глава завершается обзором HXML — формата гипермедиа, используемого Hyperview.

Состояние мобильной разработки

Прежде чем обсуждать, как применять гипермедиа на мобильных платформах, необходимо понять, как обычно строятся нативные мобильные приложения. Мы используем термин «нативный» для обозначения кода, который написан с использованием SDK, предназначенного для операционной системы смартфона (обычно Android или iOS). Этот код упаковывается в исполнительный двоичный файл, загружается и проходит проверку в магазинах приложений под управлением Google и Apple. Когда пользователи устанавливают или обновляют приложение, они скачивают этот исполняемый файл и выполняют код непосредственно в операционной системе своего устройства. Мобиль-

ные приложения в чем-то похожи на традиционные десктопные приложения для Mac, Windows или Linux. Однако между десктопными приложениями для ПК и современными мобильными приложениями существует одно важное отличие: в наши дни почти все мобильные приложения являются «сетевыми», то есть для предоставления базовой функциональности приложению требуется читать и записывать данные по интернету. Иначе говоря, сетевые мобильные приложения должны реализовывать архитектуру «клиент — сервер».

При реализации архитектуры «клиент — сервер» разработчику приходится принимать решение, как проектировать приложение — в виде тонкого или толстого клиента? Текущие мобильные экосистемы подталкивают разработчиков к решениям на основе толстых клиентов.

Почему? Вспомните: Android и iOS требуют, чтобы нативное мобильное приложение упаковывалось и распространялось как исполняемый двоичный файл. Обойти это требование невозможно. Так как разработчик должен написать код, который будет упакован в исполняемый файл, логично реализовать часть логики приложения в этом коде. С таким же успехом код может инициировать вызовы HTTP к серверу для получения данных, а затем отрендерить эти данные с использованием UI-библиотек платформы. Таким образом, разработчики логично приходят к паттерну толстого клиента, который выглядит примерно так:

- Клиент содержит код для выдачи запросов API к серверу, и код преобразует ответы в обновления UI.
- Сервер реализует HTTP API, который обменивается данными в формате JSON и мало знает о состоянии клиента.

Как и в случае со SPA в веб-среде, у этой архитектуры есть большой недостаток: логика приложения распределяется между клиентом и сервером. Иногда это означает дублирование логики (например, логики проверки данных формы). В других случаях клиент и сервер реализуют разные части общей логики приложения. Чтобы понять, что делает приложение, разработчик вынужден отслеживать взаимодействия между двумя очень разными кодовыми базами.

Существует и другой недостаток, который влияет на мобильные приложения в большей степени, чем на SPA: пересмотр API. Вспомните, что распространение и обновление приложений находится под контролем магазинов приложений. Пользователи сами решают, когда они получат (и получат ли) обновленные версии приложений. Вы как мобильный разработчик не можете опираться на то, что каждый пользователь будет работать с новейшей версией вашего приложения. Ваш код фронтенда фрагментируется между версиями, и коду бэкенда придется поддерживать их все.

Гипермедиа для мобильных приложений

Вы уже видели, что архитектура гипермедиа справляется с недостатками SPA в веб-среде. Но может ли гипермедиа работать и в мобильных приложениях? Да, может!

Как и в веб-среде, форматы гипермедиа можно использовать на мобильных устройствах и поручить им роль ядра состояния приложения. Вся логика управляется на бэкенде, а не распределяется между двумя кодовыми базами. Архитектура гипермедиа также решает неприятную проблему пересмотра API на мобильных устройствах. Так как бэкенд предоставляет ответ гипермедиа, содержащий и данные, и действия, рассинхронизация данных и UI становится невозможной. Больше не нужно беспокоиться об обратной совместимости или поддержке нескольких версий API.

Как же использовать гипермедиа для мобильных приложений? Существуют два современных подхода к применению гипермедиа для создания и поставки нативных мобильных приложений:

- веб-представления, упаковывающие проверенную веб-платформу в оболочку мобильного приложения;
- Hyperview — новая система гипермедиа, разработанная нами специально для мобильных приложений.

Веб-представления

Самый простой способ использования архитектур гипермедиа на мобильных устройствах — это привлечение существующих веб-технологий. SDK как операционной системы Android, так и iOS поддерживают «веб-представления»: браузеры без дополнительного «хрома», которые могут встраиваться в нативные приложения. Такие инструменты, как Apache Cordova, позволяют легко получить URL веб-сайта и генерируют нативные приложения iOS и Android на основе веб-представлений. Если у вас уже есть отзывчивое веб-приложение, вы получите «нативные» мобильные HDA бесплатно. Звучит слишком хорошо, чтобы быть правдой?

Конечно, у такого подхода существуют фундаментальные ограничения. Веб-платформа и мобильные платформы обладают разными возможностями, и соглашения UX для них различаются. HTML не поддерживает распространенные UI-паттерны мобильных приложений. Одно из самых больших отличий связано с тем, как на каждой платформе реализуется навигация. В веб-среде навигация основана на страницах: одна страница заменяет другую, а браузер предоставляет кнопки для прямого и обратного перехода по истории просмотра. В мобильных приложениях используется более сложная навигация, оптимизированная для физики взаимодействий на основе жестов.

- При просмотре всех запущенных приложений экраны накладываются друг на друга, образуя стеки.
- Панели вкладок в верхней или нижней части приложения позволяют переключаться между разными стеками.
- Модальные панели скользят вверх от нижнего края экрана приложения поверх других стеков и вкладок.
- В отличие от веб-страниц, мобильные страницы продолжают оставаться в памяти, рендерятся и обновляются в зависимости от состояния приложения.

В архитектуре навигации проявляется главное различие в работе мобильных и веб-приложений. Тем не менее оно не единственное. В мобильных приложениях также встречаются многие другие паттерны UX, которые не имеют нативной поддержки в веб-среде.

- Жест pull-to-refresh («потянуть, чтобы обновить») для обновления содержимого экрана.
- Горизонтальное смахивание (свайп) на элементах UI для открытия доступных действий.
- Секционированные списки с закрепленными заголовками.

Хотя эти взаимодействия не имеют встроенной поддержки в веб-браузерах, они могут моделироваться библиотеками JS. Конечно, эти библиотеки никогда не будут так же эффективны, как нативные жесты, а для их использования обычно приходится задействовать архитектуры SPA, основанные на применении JS, например React. Круг замыкается! Чтобы избежать использования типичной архитектуры толстого клиента в нативных мобильных приложениях, мы обратились к веб-представлениям. Веб-представления позволяют использовать старую добрую разметку HTML на основе гипермедиа. Но чтобы наделить мобильное приложение требуемым внешним видом и поведением, мы в конечном счете строим SPA в JS, лишаясь преимуществ технологий гипермедиа.

Если вы хотите построить мобильное приложение HDA, которое выглядит и работает как нативное, HTML будет недостаточно. Потребуется формат, спроектированный для представления взаимодействий и паттернов нативных мобильных приложений.

Именно таким является Hyperview.

Hyperview

Hyperview — система гипермедиа с открытым кодом, которая предоставляет следующие возможности.

- Формат гипермедиа для определения мобильных приложений, называемый HXML.

- Клиент для HXML, работающий на iOS и Android.
- Точки расширения HXML и клиента для настройки под конкретное приложение.

Формат

Формат HXML проектировался так, чтобы казаться знакомым веб-разработчикам, привыкшим к работе с HTML, отсюда и выбор XML как базового формата. Кроме знакомой эргономики, XML совместим с библиотеками рендеринга на стороне сервера. Например, Jinja2 идеально подходит для рендеринга HXML как библиотеки шаблонов. Знакомый формат XML и простота интеграции с бэкендом упрощают использование формата как в новых, так и в существующих кодовых базах. Посмотрите, как выглядит приложение Hello World на HXML. Этот синтаксис должен быть знаком каждому, кто работал с HTML.

Листинг 164. Hello World

```
<doc xmlns="https://hyperview.org/hyperview">
 <screen>
  <styles />
  <body>
   <header>
    <text>My first app</text>
   </header>
   <view>
    <text>Hello World!</text>
   </view>
  </body>
 </screen>
</doc>
```

Однако не стоит полагать, что HXML — просто порт HTML с другими именами тегов. В предыдущих главах вы видели, как `htmx` расширяет HTML несколькими новыми атрибутами. Эти надстройки поддерживают декларативную природу HTML, одновременно предоставляя разработчикам возможность создания полнофункциональных веб-приложений. В HXML концепции `htmx` встроены в спецификацию, а именно: HXML не ограничивается взаимодействиями «щелкнуть по ссылке» и «отправить данные формы», как базовый HTML. В нем поддерживаются разные триггеры и действия для изменения контента на экране. Эти взаимодействия объединяются в мощной концепции «вариантов поведения» (behaviors).

Разработчики даже могут определять новые действия в вариантах поведения, чтобы расширять возможности приложений без написания скриптов. Варианты поведения более подробно рассматриваются далее в этой главе.

Клиент

Hyperview предоставляет клиентскую библиотеку HXML с открытым исходным кодом, написанную в React Native. После небольшой настройки и нескольких операций в командной строке библиотека компилируется в нативные двоичные файлы приложений для iOS или Android. Пользователи устанавливают приложение на своем устройстве через магазин приложений. При запуске приложение выдает запрос HTTP к настроенному URL, а ответ HXML рендерится как первый экран приложения.

Может показаться немного странным, что разработка HDA с использованием Hyperview требует клиентского двоичного файла, предназначенного для единственной цели. В конце концов, мы не предлагаем пользователям для просмотра веб-приложения сначала загрузить и установить двоичный файл. Нет, пользователь просто вводит URL в адресной строке обычного браузера. Один клиент HTML рендерит приложения от любого сервера HTML.

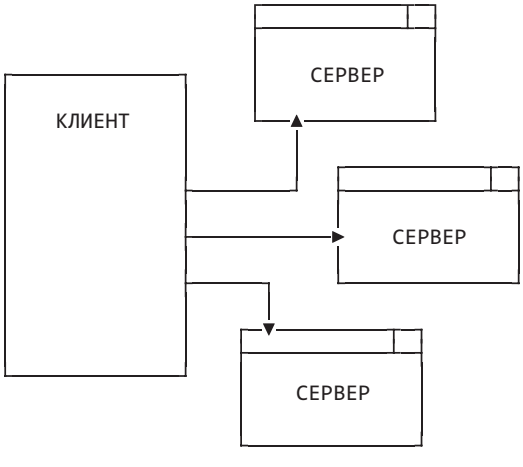


Рис. 11. Один клиент HTML, несколько серверов HTML

Теоретически можно построить эквивалентный «обычный браузер Hyperview». Такой клиент HXML сможет рендерить приложения от любого сервера HXML, а пользователь будет вводить URL, чтобы выбрать приложение, которое он хочет использовать. Однако iOS и Android строятся на основе концепции одноцелевых приложений. Предполагается, что пользователь находит и устанавливает приложения из магазина и запускает их с домашнего экрана своего устройства. Hyperview принимает эту концепцию популярных мобильных платформ, в которых центральное место занимают приложения. Это означает, что клиент

HXML (двоичный файл приложения) рендерит свой пользовательский интерфейс, полученный от одного заранее настроенного сервера HXML:

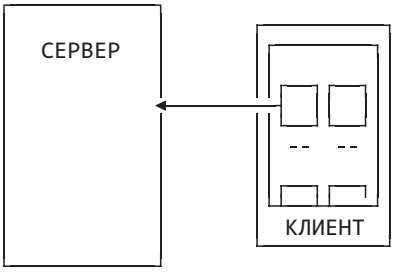


Рис. 12. Один клиент HXML, один сервер HXML

К счастью, разработчикам не нужно писать клиент HXML с нуля; клиентская библиотека с открытым кодом выполняет 99 % работы. И как будет показано в следующем разделе, управление как клиентом, так и сервером из HDA имеет целый ряд преимуществ.

Расширяемость

Чтобы понять преимущества архитектуры Hyperview, необходимо сначала обсудить недостатки веб-архитектуры. В веб-среде любой браузер может рендерить HTML от любого веб-сервера. Такой уровень совместимости может достигаться только при наличии четко определенных стандартов, таких как HTML5. Однако определение и эволюция стандартов — трудоемкий процесс. Например, комитету W3C понадобилось более семи лет для перехода от первого черновика до рекомендаций по спецификациям HTML5. И это неудивительно, если подумать, насколько тщательно должны продумываться изменения, затрагивающие такое количество людей. Но это означает, что прогресс движется медленно. Веб-разработчикам приходится годами ждать, пока в браузерах появится широкая поддержка необходимой функциональности.

Каковы же преимущества архитектуры Hyperview? В мобильных приложениях Hyperview рендерится только разметка HXML от сервера. Не нужно беспокоиться о совместимости между сервером и другими мобильными приложениями или между мобильным приложением и другими серверами. Не нужно обращаться в комитеты по стандартизации за консультациями. Если вы хотите добавить в свое мобильное приложение функциональность мигания, добавьте в клиент элемент `<blink>` и начните возвращать элементы `<blink>` в ответах HXML сервера. Собственно, клиентская библиотека Hyperview строилась с учетом подобной расширяемости. Существуют точки расширения для нестандартных элементов UI и нестандартных действий поведения. Мы

всячески рекомендуем разработчикам использовать эти расширения, чтобы сделать HXML более выразительным и адаптированным к функциональности приложений.

А с расширением формата HXML и самого клиента у Hyperview отпадает необходимость включения в HXML уровня скриптов. Функциональность, требующая логики на стороне клиента, «встраивается» в клиентский двоичный файл. Ответы HXML остаются «чистыми», а UI и взаимодействия представляются в декларативном XML.

Какую архитектуру гипермедиа использовать?

Мы обсудили два подхода к мобильной разработке с использованием систем гипермедиа:

- создание бэкенда, возвращающего HTML, и поставка HTML в мобильном приложении через веб-представление;
- создание бэкенда, возвращающего HXML, и поставка HXML в мобильном приложении с клиентом Hyperview.

Мы намеренно сформулировали два описания именно так, чтобы подчеркнуть отличия между ними. В конце концов, оба решения базируются на системах гипермедиа, только с разными форматами и клиентами. Оба варианта решают фундаментальные проблемы традиционным путем, сходным с SPA-подобной мобильной разработкой:

- бэкенд управляет полным состоянием приложения;
- вся логика приложения сосредоточена в одном месте.

Приложение всегда выполняется в последней версии, и исчезают проблемы с пересмотром API.

Какой вариант использовать для мобильных HDA? По собственному опыту создания обоих видов приложений мы считаем, что подход с Hyperview улучшает качество взаимодействия с пользователем. Веб-представления всегда кажутся чужеродными в iOS и Android; просто не существует хорошего способа реализовать паттерны навигации и взаимодействия, которые рассчитывает видеть пользователь мобильного устройства. Технология Hyperview создавалась специально для преодоления ограничений решений с толстым клиентом и веб-представлениями. После исходных вложений в изучение Hyperview вы сможете пользоваться всеми преимуществами архитектуры гипермедиа, не снижая качества взаимодействия с пользователем.

Конечно, если у вас уже есть простое, адаптированное для мобильных устройств веб-приложение, то выбор решения с веб-представлением представляется разум-

ным. Вы безусловно сэкономите время за счет того, что вам не придется поставлять свое приложение в формате HXML, помимо HTML. Хотя, как будет показано в конце этой главы, преобразование существующего гипермедиа-управляемого веб-приложения в мобильное приложение Hyperview не требует большого объема работы. Но прежде чем мы доберемся до этого, необходимо ввести концепции элементов и поведений в Hyperview. Тогда мы перестроим свое приложение для управления контактами в Hyperview.

КОГДА НЕ СТОИТ ИСПОЛЬЗОВАТЬ ГИПЕРМЕДИА ДЛЯ ПОСТРОЕНИЯ МОБИЛЬНОГО ПРИЛОЖЕНИЯ?

Гипермедиа не всегда бывает лучшим вариантом для создания мобильных приложений. Как и в веб-среде, приложения, требующие динамических пользовательских интерфейсов (например, приложение электронной таблицы), лучше реализуются в коде на стороне клиента. Кроме того, некоторые приложения должны работать в полностью автономном режиме. Так как приложениям HDA необходим сервер для рендеринга UI, эта архитектура вряд ли хорошо подойдет для офлайновых мобильных приложений.

Но как и в веб-среде, разработчики могут применять гибридный подход для построения мобильных приложений. Для динамических экранов можно использовать сложную логику на стороне клиента, тогда как менее динамические экраны можно создавать на основе веб-представлений или Hyperview. Таким образом, разработчики могут расходовать свой *бюджет сложности* на ядро приложения, оставляя простые экраны простыми.

Знакомство с HXML

Hello World!

Формат HXML проектировался так, чтобы казаться естественным для веб-разработчиков с опытом HTML. Рассмотрим поближе приложение Hello World, созданное на HXML.

Листинг 165. Приложение Hello World

```
<doc xmlns="https://hyperview.org/hyperview"> ❶
 <screen> ❷
  <styles />
  <body> ❸
   <header> ❹
    <text>My first app</text>
   </header>
   <view> ❺
    <text>Hello World!</text> ❻
   </view>
```

```
  </body>
 </screen>
</doc>
```

❶ Корневой элемент приложения HXML.
❷ Элемент, представляющий экран приложения.
❸ Элемент, представляющий пользовательский интерфейс экрана.
❹ Элемент, представляющий заголовок экрана.
❺ Элемент-обертка для контента, выводимого на экран.
❻ Текстовый контент, выводимый на экран.

Выглядит логично, не так ли? Как и с HTML, синтаксис определяет дерево элементов, использующих начальный (`<screen>`) и конечный (`</screen>`) теги. Элементы могут содержать другие элементы (`<view>`) или текст (`Hello World!`). Элементы также могут быть пустыми, представляемыми пустыми тегами (`<styles />`). Тем не менее можно заметить, что имена элементов HXML отличаются от имен элементов HTML. Посмотрим внимательнее на каждый из этих элементов, чтобы понять, что они делают.

- `<doc>` — корень приложения HXML. Считайте, что это аналог элемента `<html>` в HTML. Обратите внимание: элемент `<doc>` содержит атрибут `xmlns="https://hyperview.org/hyperview"`, определяющий пространство имен по умолчанию для документа. Пространства имен — функциональность XML, которая позволяет одному документу содержать элементы, определяемые разными разработчиками. Чтобы предотвратить возможные конфликты, когда два разработчика используют одно имя для своего элемента, каждый разработчик определяет уникальное пространство имен. Пространства имен будут более подробно рассмотрены позже в этой главе, когда мы будем обсуждать нестандартные элементы и поведения. А пока достаточно знать, что элементы в документе HXML без явно указанного пространства имен считаются частью пространства имен https://hyperview.org/hyperview.
- `<screen>` представляет пользовательский интерфейс, который рендерится на одном экране мобильного приложения. Один элемент `<doc>` может содержать несколько элементов `<screen>`, но об этом позже. Обычно элемент `<screen>` содержит элементы, определяющие контент и стилевое оформление экрана.
- `<styles>` задает стили пользовательского интерфейса на экране. Стилевое оформление Hyperview в этой главе подробно не рассматривается. Отметим только, что, в отличие от HTML, Hyperview не использует отдельный язык (CSS) для определения стилей. Вместо этого правила стилевого оформления (цвета, интервалы, макет, шрифты) определяются в HXML. Эти правила затем явно указываются в элементах UI, по аналогии с использованием классов в CSS.

- `<body>` определяет фактический пользовательский интерфейс экрана. Этот элемент включает весь текст, графику, кнопки, формы и т. д., которые будут отображаться для пользователя. Элемент эквивалентен элементу `<body>` в HTML.
- `<header>` определяет заголовок экрана. Обычно в мобильных приложениях заголовок включает некоторые средства навигации (например, кнопку возврата) и текст названия экрана. Заголовок полезно определять отдельно от тела документа. Некоторые мобильные ОС используют разные переходы для заголовка и остального контента экрана.
- `<view>` — основной структурный элемент макета и структуры тела документа. Можно считать его аналогом `<div>` в HTML. Учтите, что, в отличие от HTML, `<div>` не может содержать текст.
- Элементы `<text>` — единственный способ рендеринга текста в пользовательском интерфейсе. В приведенном примере текст Hello World содержится в элементе `<text>`.

И это все, что нужно для написания простейшего приложения Hello World в HXML. Конечно, это не самый интересный пример. Рассмотрим некоторые встроенные экранные элементы.

Элементы UI

Списки

Один из самых распространенных мобильных паттернов — прокрутка списка. Из-за физических свойств экрана телефона (вытянут по вертикали) и интуитивно понятного жеста смахивания большим пальцем вверх и вниз этот паттерн хорошо подходит для многих экранов.

HXML содержит специализированные элементы для представления списков.

Листинг 166. Элемент list

```
<list> ❶
 <item key="item1"> ❷
  <text>Первый элемент списка</text> ❸
 </item>
 <item key="item2">
  <text>Второй элемент списка</text>
 </item>
</list>
```

❶ Элемент, представляющий список.

❷ Элемент, представляющий отдельный пункт списка, имеет уникальный ключ.

❸ Содержимое пункта списка.

Списки представляются двумя новыми элементами. `<list>` инкапсулирует все пункты списка. К нему можно применять стилевое оформление как к обобщенному `<view>` (ширина, высота и т. д.). Элемент `<list>` содержит только элементы `<item>`. Разумеется, они представляют уникальные пункты списка. Обратите внимание: элемент `<item>` должен содержать атрибут `key`, уникальный среди всех пунктов списка.

Возможно, вы спросите, для чего нужен нестандартный синтаксис для пунктов списка? Разве нельзя просто использовать набор элементов `<view>`? Да, для списков с небольшим количеством пунктов использование вложенных элементов `<view>` работает неплохо. Тем не менее при достаточно большом количестве пунктов может потребоваться оптимизация для поддержки плавной прокрутки. Представьте просмотр ленты постов в приложении социальной сети. В процессе прокрутки нередко встречаются ленты с сотнями и даже тысячами постов. В любой момент список можно прокрутить почти к любой позиции. Память мобильных устройств обычно ограниченна. На хранение полностью отрендеренного списка пунктов в памяти могут потребоваться ресурсы, превышающие доступные. Вот почему iOS и Android предоставляют API для оптимизированных списков: эти API знают, какая часть списка находится на экране. Для экономии памяти пункты списка, которые пока не видны, скрываются, а объекты пунктов перерабатываются. При использовании элементов `<list>` и `<item>` в HXML клиент Hyperview умеет использовать API оптимизированных списков, чтобы приложение работало более эффективно.

Также стоит упомянуть о поддержке секционированных списков в HXML. Секционированные списки хорошо подходят для UI на основе списков, в которых пункты могут группироваться для удобства пользователя. Например, в пользовательском интерфейсе для вывода меню ресторана блюда могут группироваться по типу.

Листинг 167. Элемент секционированного списка

```
<section-list> ❶
 <section> ❷
  <section-title> ❸
   <text>Закуски</text>
  </section-title>
  <item key="1"> ❹
   <text>Картофель фри</text>
  </item>
  <item key="2">
   <text>Луковые кольца</text>
  </item>
 </section>

 <section> ❺
  <section-title>
```

```
   <text>Основные блюда</text>
  </section-title>
  <item key="3">
   <text>Бургер</text>
  </item>
 </section>
</section-list>
```

❶ Элемент, представляющий список с разделами.
❷ Первый раздел меню.
❸ Элемент названия раздела рендерит текст «Закуски».
❹ Элемент, представляющий пункт меню.
❺ Второй раздел меню.

Можно заметить несколько отличий между `<list>` и `<section-list>`. Элемент секционированного списка содержит только элементы `<section>`, представляющие группы пунктов списка. Секция может содержать элемент `<section-title>`, который используется для рендеринга UI, служащего заголовком списка. Этот заголовок «закреплен», то есть остается на экране во время прокрутки пунктов списка соответствующей секции. Наконец, элементы `<item>` работают так же, как в обычных списках, но могут содержаться только в `<section>`.

Изображения

Работа с изображениями в Hyperview ведется практически так же, как в HTML, но и здесь существует ряд различий.

Листинг 168. Элемент Image

```
<image source="/profiles/1.jpg" style="avatar" />
```

Атрибут `source` указывает, откуда должно загружаться изображение. Как и в HTML, источник может определяться как абсолютным, так и относительным URL. Кроме того, источник может содержать закодированный URI данных, например ``. Однако можно использовать и «локальный» URL, ссылающийся на изображение, упакованное в мобильное приложение в виде ресурса. Локальный URL снабжается префиксом `./`.

Листинг 169. Элемент Image со ссылкой на локальный источник

```
<image source="./logo.png" style="logo" />
```

Использование локальных URL является оптимизацией. Так как изображения хранятся на мобильном устройстве, они не требуют сетевых запросов и быстро появляются на экране. С другой стороны, упаковка изображения в двоичный файл мобильного приложения увеличивает размер двоичного файла. Исполь-

зование локальных изображений станет хорошим компромиссом для изображений, к которым приложения часто обращается, но которые редко изменяются. Хорошие примеры — логотип приложения или стандартные значки на кнопках.

Также отметим наличие атрибута `style` у элемента `<image>`. В HXML изображениям обязательно должен быть назначен стиль с указанием ширины и высоты изображения. В этом отличие от HTML, где элементам `<img>` не нужно явно задавать ширину и высоту. Веб-браузеры проводят повторную оценку контента веб-страницы после получения изображения, когда станут известны его размеры. Хотя пересчет контента может быть обоснован для веб-документов, пользователи не ожидают, что в мобильных приложениях при загрузке контента будет пересчитываться макет. Чтобы использовать статический макет, не меняющийся при пересчете, HXML требует, чтобы размеры были известны до загрузки изображения.

Поля ввода

О Hyperview можно рассказывать очень долго. Поскольку эта глава задумывалась как краткий обзор, а не подробный справочник, мы остановимся лишь на нескольких типах полей ввода. Начнем с простейшего из них — текстовых полей.

Листинг 170. Элемент текстового поля

```
<text-field
 name="first_name" ❶
 style="input" ❷
 value="Адам" ❸
 placeholder="Имя" ❹
/>
```

❶ Имя, используемое при сериализации данных от этого поля ввода.
❷ Класс стиля, применяемый к элементу UI.
❸ Текущее значение, заданное для поля.
❹ Заполнитель, который должен выводиться при пустом значении.

Этот элемент хорошо знаком каждому, кто создавал текстовые поля в HTML. Одно из различий заключается в том, что большинство полей ввода в HTML использует элемент `<input>` с атрибутом `type`, например `<input type="text">`. В Hyperview каждое поле ввода обладает уникальным именем, в данном случае `<text-field>`. Используя разные имена, можно создавать более выразительную разметку XML для представления полей ввода.

Предположим, вы рендерите пользовательский интерфейс для выбора одного из нескольких вариантов. В HTML для этого использовалось бы поле ввода типа `radio`, например `<input type="radio" name="choice" value="option1" />`.

Каждый вариант представлен уникальным элементом `input`. Нам такое решение никогда не казалось идеальным. В большинстве случаев радиокнопки группируются с одним именем. Подход HTML ведет к увеличению объема шаблонного кода (дублирование `type="radio"` и `name="choice"` для каждого варианта). Кроме того, мобильные ОС не предлагают сильного стандартного UI для выбора одного варианта, такого как радиокнопки в десктопных приложениях. В большинстве мобильных приложений используются расширенные, специализированные UI для таких взаимодействий. По этой причине в HXML для реализации этого интерфейса существует элемент с именем `<select-single>`.

Листинг 171. Элемент для выбора одного варианта

```
<select-single name="choice"> ❶
 <option value="option1"> ❷
  <text>Вариант 1</text> ❸
 </option>
 <option value="option2">
  <text>Вариант 2</text>
 </option>
</select-single>
```

❶ Элемент, представляющий поле ввода для выбора одного варианта. Имя выбранного варианта определяется один раз.

❷ Элемент, представляющий один из вариантов. Здесь определяется выбранное значение.

❸ Пользовательский интерфейс выбора. В данном случае text, но могут использоваться любые элементы UI.

Элемент `<select-single>` — родитель для поля ввода, предназначенного для выбора одного варианта из многих. Этот элемент содержит атрибут `name`, используемый при сериализации выбранного варианта. Элементы `<option>` внутри `<select-single>` представляют доступные варианты. Обратите внимание: у каждого элемента `<option>` есть атрибут `value`. При нажатии он принимает выбранное значение для данного поля ввода. Элемент `<option>` может содержать любые другие элементы UI. Это означает, что разработчик не ограничен рендерингом поля ввода в виде списка радиокнопок с ярлыками. Можно использовать радиокнопки, теги, изображения и вообще все, что будет интуитивно уместно для интерфейса. Стили HXML поддерживают модификаторы для нажатого и выбранного состояния, что позволяет настроить интерфейс, чтобы выделить выбранный вариант.

На описание всех возможностей полей ввода в HXML потребовалась бы целая глава. Мы ограничимся примерами других полей ввода с кратким обзором их функциональности.

- Элемент `<select-multiple>` работает как `<select-single>`, но поддерживает возможность переключения состояния нескольких вариантов. Он заменяет поля ввода `checkbox` в HTML.
- Элемент `<switch>` рендерит переключатель «вкл/выкл», часто используемый в мобильных UI.
- Элемент `<date-field>` поддерживает ввод конкретных дат; в нем есть ряд настроек форматирования, диапазонов значений и т. д.

И еще два замечания, касающиеся полей ввода. Элемент `<form>` используется для группировки полей для сериализации. Когда пользователь выполняет действие, инициирующее запрос на бэкенд, клиент Hyperview сериализует все поля ввода окружающего элемента `<form>` и включает их в запрос. Это относится к запросам `GET` и `POST`. Мы рассмотрим эту особенность более подробно при обсуждении поведений далее в этой главе.

Кроме того, мы поговорим о поддержке нестандартных элементов в HXML. С нестандартными элементами также можно создавать собственные элементы ввода. Они открывают возможность реализации невероятно мощных взаимодействий с простым синтаксисом XML, хорошо интегрируемым с остальными средствами HXML.

Стили

Мы еще не говорили о том, как применять стили ко всем элементам HXML. Из приложения Hello World вы узнали, что каждый элемент `<screen>` может содержать элемент `<styles>`. Вернемся к приложению Hello World и заполним элемент `<styles>`.

Листинг 172. Пример стилевого оформления UI

```
<doc xmlns="https://hyperview.org/hyperview">
 <screen>
  <styles> ❶
   <style class="body" flex="1" flexDirection="column" /> ❷
   <style class="header" borderBottomWidth="1" borderBottomColor="#ccc" />
   <style class="main" margin="24" />
   <style class="h1" fontSize="32" />
   <style class="info" color="blue" />
  </styles>

  <body style="body"> ❸
   <header style="header">
    <text style="info">Мое первое приложение</text>
   </header>
   <view style="main">
    <text style="h1 info">Hello World!</text> ❹
   </view>
```

```
  </body>
 </screen>
</doc>
```

❶ Элемент, инкапсулирующий все стили экрана.
❷ Пример определения класса стиля для body.
❸ Применение класса стиля body к элементу UI.
❹ Пример применения к элементу нескольких классов стилей (h1 и info).

Заметим, что в HXML стили являются частью формата XML, а не отдельным языком, как CSS. Тем не менее можно провести некоторые параллели между правилами CSS и элементом `<style>`. Правило CSS состоит из селектора и объявлений. В текущей версии HXML единственным доступным селектором является имя класса, заданное атрибутом `class`. Остальные атрибуты элемента `<style>` составляют объявления, состоящие из свойств и их значений.

Элементы UI внутри `<screen>` могут ссылаться на правила `<style>`, добавляя имена классов в свойство `<style>`. Обратите внимание: элемент `<text>`, в который заключен элемент «Hello World!», ссылается на два класса стилей: `h1` и `info`. Стили из соответствующих классов объединяются в порядке их следования в элементе. Стоит заметить, что свойства стилей похожи на свойства в CSS (цвет, поля/заполнители, границы и т. д.). Пока доступно одно ядро построения макета на основе flexbox.

Правила стилей могут быть довольно подробными. Для краткости мы не будем включать элемент `<styles>` в остальные примеры этой главы, если в этом не будет необходимости.

Нестандартные (кастомные) элементы

Базовые элементы UI, поставляемые вместе с Hyperview, относительно просты. Многие мобильные приложения требуют более мощных элементов для реализации более качественного взаимодействия с пользователем. К счастью, HXML позволяет легко внедрить нестандартные элементы в свой синтаксис. Дело в том, что HXML на самом деле представляет собой обычный XML, то есть «расширяемый язык разметки» (eXtensible Markup Language). Расширяемость уже встроена в этот формат! Разработчику ничто не мешает определять новые элементы и атрибуты для представления нестандартных элементов.

Рассмотрим эту возможность на конкретном примере. Предположим, в приложение «Hello World» требуется добавить элемент с географической картой. На карте должна определяться конкретная область с одним или несколькими маркерами. Переведем эти требования в формат XML.

- Элемент `<area>` представляет область, отображаемую на карте. Для определения области в элемент включаются атрибуты широты (`latitude`) и долготы (`longitude`) центра области, а также атрибуты `latitude-delta` и `longitude-delta`, определяющие размеры отображаемой области относительно центра.
- Элемент `<marker>` представляет маркер в этой области. Координаты маркера определяются его атрибутами `latitude` и `longitude`.

С этими нестандартными элементами XML экземпляр карты в приложении может выглядеть так:

Листинг 173. Нестандартные элементы в HXML

```
<doc xmlns="https://hyperview.org/hyperview">
 <screen>
  <body>
   <view>
    <text>Hello World!</text>
    <area latitude="37.8270" longitude="122.4230" latitude-delta="0.1"
longitude-delta="0.1"> ❶
     <marker latitude="37.8118" longitude="-122.4177" /> ❷
    </area>
   </view>
  </body>
 </screen>
</doc>
```

❶ Нестандартный элемент, представляющий область, которая отображается на карте.

❷ Нестандартный элемент, представляющий маркер с указанными координатами на карте.

Синтаксис выглядит абсолютно естественно на фоне базовых элементов HXML. Однако здесь кроется потенциальная проблема. «Area» и «marker» — слишком общие имена. Например, элементы `<area>` и `<marker>` вполне могут использоваться при построении диаграмм и графиков. Если в приложении присутствуют как карты, так и диаграммы, разметка HXML становится неоднозначной. Что должен отрендерить клиент, когда он встречает элемент `<area>` или `<marker>`?

На помощь приходят пространства имен XML. Они полностью устраняют неоднозначность и потенциальные конфликты между элементами и атрибутами, используемыми для представления разных сущностей. Напомним, что элемент `<doc>` объявляет https://hyperview.org/hyperview пространством имен по умолчанию для всего документа. Так как никакие другие элементы не определяют пространства имен, каждый элемент в приведенном примере является частью пространства имен https://hyperview.org/hyperview.

Определим новое пространство имен для элементов карты. Так как оно не будет использоваться по умолчанию для документа, также необходимо указать его в префиксе, добавляемом к элементам:

```
<doc xmlns="https://hyperview.org/hyperview"
xmlns:map="https://mycompany.com/hyperview-map">
```

Новый атрибут объявляет, что префикс `map:` связан с пространством имен `"https://mycompany.com/hyperview-map"`. Пространству имен можно назначить любое имя, но помните, что имя должно быть уникальным, чтобы избежать конфликтов. Для обеспечения уникальности можно воспользоваться доменом компании/приложения. Теперь, когда у вас есть пространство имен и префикс, их можно применить к элементам:

Листинг 174. Назначение пространств имен нестандартным элементам

```
<doc xmlns="https://hyperview.org/hyperview"
xmlns:map="https://mycompany.com/hyperview-map"> ❶
 <screen>
  <body>
   <view>
    <text>Hello World!</text>
    <map:area latitude="37.8270" longitude="122.4230" latitude-
delta="0.1" longitude-delta="0.1"> ❷
     <map:marker latitude="37.8118" longitude="-122.4177" /> ❸
    </map:area> ❹
   </view>
  </body>
 </screen>
</doc>
```

❶ Определение пространства имен с псевдонимом map.
❷ Добавление пространства имен в начальный тег area.
❸ Добавление пространства имен в самозакрывающийся тег marker.
❹ Добавление пространства имен к завершающему тегу area.

И это все! Если бы мы разрабатывали также специализированную библиотеку для построения диаграмм с элементами `area` и `marker`, можно было бы создать уникальное пространство имен и для этих элементов. В документе HXML было бы легко отличить `<map:area>` от `<chart:area>`.

В этот момент возникает логичный вопрос: откуда клиент Hyperview знает, что нужно отрендерить карту, если в документ включается элемент `<map:area>`? Верно, пока мы только определили формат нестандартного элемента, но не реализовали элемент в составе функциональности приложения. Реализация нестандартных элементов будет рассмотрена в следующей главе.

Поведение

Как отмечалось ранее, HTML поддерживает два базовых типа взаимодействий:

- щелчок на гиперссылке: клиент выдает запрос `GET` и рендерит ответ в виде новой веб-страницы;
- отправка данных формы: клиент (обычно) выдает запрос `POST` с сериализованным контентом формы и рендерит ответ как новую веб-страницу.

Щелчков на гиперссылках и отправки данных форм достаточно для построения простых веб-приложений. Тем не менее, если мы ограничимся этими двумя взаимодействиями, мы не сможем строить более функциональные UI. Что, если нам необходимо, чтобы действие происходило при наведении указателя мыши на конкретный элемент или при вхождении контента в область просмотра при прокрутке? С базовым HTML это сделать невозможно. Кроме того, щелчки и отправки данных формы приводят к загрузке полной новой веб-страницы. А если требуется обновить только небольшую часть текущей страницы? Это типичный сценарий во многих полнофункциональных веб-приложениях, когда пользователи ожидают загрузки и обновления контента без перехода к новой странице.

Итак, с базовым HTML взаимодействия (щелчки и отправки данных) ограниченны и тесно связаны с одним действием (загрузкой новой страницы). Конечно, используя JavaScript, можно расширить HTML и добавить синтаксис поддержки нужных взаимодействий. Для этого `htmx` использует новый набор атрибутов.

- Взаимодействия могут добавляться к любым элементам, не только к ссылкам и формам.
- Взаимодействия могут инициироваться по щелчку, отправке данных, наведению указателя мыши или любому другому событию JavaScript.
- Действия, происходящие из-за триггера, могут изменять текущую страницу, а не только запрашивать новую.

За счет разделения элементов, триггеров и действий в htmx можно строить богатые гипермедиа-управляемые приложения способом, тесно совместимым с синтаксисом HTML и серверной веб-разработкой.

HXML берет идею определения взаимодействий через триггеры и действия и встраивает их в спецификацию. Такие взаимодействия называются поведениями. Для их определения используется специальный элемент `<behavior>`. Пример простого поведения, которое проталкивает новый мобильный экран в стек навигации.

Листинг 175. Базовое поведение

```
<text>
 <behavior ❶
  trigger="press" ❷
```

```
  action="push" ❸
  href="/next-screen" ❹
 />
 Нажми меня!
</text>
```

❶ Элемент, инкапсулирующий взаимодействие с родительским элементом <text>.
❷ Триггер, который выполняет взаимодействие, — в данном случае нажатие (press) элемента <text>.
❸ Действие, которое выполняется при выполнении условия, — в данном случае проталкивание (push) нового экрана в текущий стек.
❹ href для загрузки нового экрана.

Проанализируем происходящее в этом примере. Сначала определяется элемент `<text>` с контентом `Нажми меня!`. Элементы `<text>` уже приводились в примерах HXML, так что ничего нового здесь нет. Но сейчас элемент `<text>` содержит новый дочерний элемент `<behavior>`. Этот элемент определяет взаимодействие с родительским элементом `<text>`. Он содержит два атрибута, необходимых для любого поведения:

- `trigger`: определяет действие пользователя, которое инициирует поведение;
- `action`: определяет, что происходит при инициировании.

В этом примере атрибуту `trigger` присваивается значение `press`, означающее, что это взаимодействие происходит при нажатии на элемент `<text>`. Атрибуту `action` присваивается значение `push` — действие, которое проталкивает новый экран в стек навигации. Наконец, Hyperview необходимо знать, какой контент должен загружаться для проталкиваемого экрана. В этом поможет атрибут `href`. Заметим, что определять полный URL не нужно. По аналогии с HTML атрибуту `href` может быть присвоен как абсолютный, так и относительный URL.

Ниже приведен первый пример поведения в HXML. Возможно, вам покажется, что синтаксис слишком объемный. В самом деле, нажатие на элемент для перехода к новому экрану является одним из самых популярных взаимодействий в мобильном приложении. Было бы хорошо иметь более простой синтаксис для распространенного случая. К счастью, атрибуты `trigger` и `action` имеют значения по умолчанию `press` и `push` соответственно. Следовательно, их можно опустить, чтобы оптимизировать синтаксис.

Листинг 176. Базовое поведение со значениями по умолчанию

```
<text>
 <behavior href="/next-screen" /> ❶
 Нажми меня!
</text>
```

❶ При нажатии это поведение открывает новый экран с заданным URL.

Разметка `<behavior>` генерирует такое же взаимодействие, как в предыдущем примере. С атрибутами по умолчанию элемент `<behavior>` похож на якорный тег `<a>` в HTML. Однако полный синтаксис достигает цели по ослаблению связей между элементами, триггерами и действиями.

- Поведения можно добавлять к любому элементу, они не ограничиваются ссылками и формами.
- Поведения могут явно задавать атрибут `trigger`, не только щелчки или отправки данных формы.
- Поведения могут задавать явное действие, не только запрос новой страницы.
- Дополнительные атрибуты, такие как `href`, предоставляют больше контекста для действия.

Кроме того, использование специализированного элемента `<behavior>` означает, что один элемент может определять несколько поведений. Это позволяет выполнять сразу несколько действий по одному триггеру или же разные действия по разным триггерам от одного элемента. Примеры множественного поведения будут приведены в конце главы, а сначала необходимо продемонстрировать разнообразие поддерживаемых действий и триггеров.

Действия

Действия поведения в Hyperview делятся на четыре категории:

- действия навигации, которые загружают новые экраны и перемещаются между ними;
- действия обновления, которые изменяют HXML текущего экрана;
- системные действия, которые взаимодействуют с функциональностью уровня ОС;
- нестандартные действия, которые выполняют произвольный код, включенный в клиент.

Действия навигации

Простейший тип действий — `push` — уже был продемонстрирован выше. Действие `push` классифицируется как «действие навигации», так как оно связано с навигацией по экранам мобильного приложения. Проталкивание экрана в навигационный стек — всего лишь одно из нескольких действий навигации, поддерживаемых в Hyperview. Пользователи также должны иметь возможность возвращаться к предыдущим экранам, открывать и закрывать модальные окна, переключаться между вкладками или переходить к экранам по своему выбору. Все эти типы навигации поддерживаются разными значениями атрибута `action`:

- `push` — проталкивает новый экран в текущий стек навигации; новый экран выскальзывает справа и размещается поверх текущего экрана;
- `new` — открывает новый стек навигации в модальном режиме; экран выскальзывает снизу и размещается поверх текущего экрана;
- `back` — действие, дополнительное по отношению к `push`, извлекает текущий экран из стека навигации (экран скользит направо);
- `close` — действие, дополнительное по отношению к `new`; закрывает текущий стек навигации (экран скользит вниз);
- `reload` — напоминает кнопку обновления в браузере, но заново запрашивает контент текущего экрана;
- `navigate` — действие ищет экран с заданным атрибутом `href`, уже загруженный в приложении. Если экран существует, то приложение переходит к нему. Если экран не существует, действие работает так же, как `push`.

Действия `push`, `new` и `navigate` загружают новый экран. Соответственно, им необходим атрибут `href`, чтобы клиент Hyperview знал, какой контент следует запросить для нового экрана. `back` и `close` не загружают новые экраны, поэтому атрибут `href` им не нужен. `reload` — интересный случай. По умолчанию действие использует URL экрана при повторном запросе контента этого экрана. Но если вы хотите заменить экран другим, предоставьте атрибут `href` с перезагрузкой `reload` на элементе `behavior`.

Рассмотрим приложение, в котором используется несколько действий навигации на одном экране.

Листинг 177. Примеры действий навигации

```
<screen>
 <body>
  <header>
   <text>
    <behavior action="back" /> ❶
    Back
   </text>

   <text>
    <behavior action="new" href="/widgets/new" /> ❷
    New Widget
   </text>
  </header>
  <text>
   <behavior action="reload" /> ❸
   Check for new widgets
  </text>
  <list>
   <item key="widget1">
```

```
    <behavior action="push" href="/widgets/1" /> ❹
   </item>
  </list>
 </body>
</screen>
```

❶ Переводит пользователя к предыдущему экрану.

❷ Открывает новое модальное окно для добавления виджета.

❸ Перезагружает содержимое экрана с отображением новых виджетов, полученных на бэкенд.

❹ Проталкивает новый экран с подробной информацией о заданном виджете.

На многих экранах приложения требуется функция возвращения к предыдущему экрану. Обычно она реализуется кнопкой в заголовке, который использует действие `back` или `close` в зависимости от того, как был открыт экран. В нашем примере мы предполагаем, что экран виджетов был занесен в стек навигации, так что действие `back` подойдет. Заголовок содержит вторую кнопку, которая позволяет пользователю ввести данные для нового виджета. Нажатие этой кнопки открывает модальное окно с экраном New Widget (Новый виджет). Так как этот экран New Widget открывается в модальном режиме, необходимо соответствующее действие `close`, которое закроет его, и на экране снова появится экран виджетов. Наконец, для получения подробной информации о заданном виджете каждый элемент `<item>` содержит поведение с действием `push`. Это действие проталкивает экран Widget Detail (Информация о виджете) в текущий стек навигации. Как и в случае с экраном виджетов, в заголовке экрана Widget Detail должна присутствовать кнопка, использующая действие `back` для возврата к предыдущему экрану.

В веб-среде браузер обрабатывает базовые потребности навигации — переходы назад/вперед, перезагрузку текущей страницы или переход к закладке. iOS и Android не предоставляют подобную универсальную навигацию для нативных мобильных приложений. Разработчики приложений должны сами позаботиться об этом. Действия навигации в HXML предоставляют разработчикам простой, но мощный механизм для построения архитектуры, подходящий для их приложений.

Действия обновления

Действия поведения не ограничиваются навигацией между экранами. Они также могут использоваться для изменения контента на текущем экране. Мы называем это «действиями обновления». Как и в случае с действиями навигации, действия обновления выдают запрос на бэкенд. Однако ответ содержит не полный документ HXML, а фрагмент HXML. Этот фрагмент добавляется в HXML текущего экрана, что приводит к обновлению UI. Атрибут `action` элемента

`<behavior>` определяет, как фрагмент встраивается в HXML. Также необходимо включить в `<behavior>` новый атрибут `target` для определения места, в котором фрагмент встраивается в существующий документ. Атрибут `target` содержит ссылку на идентификатор существующего элемента на экране.

Hyperview поддерживает следующие действия обновления, представляющие разные способы добавления фрагмента на экран:

- `replace` — заменяет весь целевой элемент фрагментом;
- `replace-inner` — заменяет дочерние элементы целевого элемента фрагментом;
- `append` — добавляет фрагмент после последнего дочернего элемента в целевом элементе;
- `prepend` — добавляет фрагмент перед первым дочерним элементом целевого элемента.

Рассмотрим несколько примеров, которые помогут наглядно представить сказанное. Для этих примеров предположим, что бэкенд принимает запросы `GET` к `/fragment`, а ответ содержит фрагмент HXML вида `<text>My fragment</text>`.

Листинг 178. Примеры действий обновления

```
<screen>
 <body>
  <text>
   <behavior action="replace" href="/fragment" target="area1" /> ❶
   Replace
  </text>
  <view id="area1">
   <text>Existing content</text>
  </view>

  <text>
   <behavior action="replace-inner" href="/fragment" target="area2" /> ❷
   Replace-inner
  </text>
  <view id="area2">
   <text>Existing content</text>
  </view>

  <text>
   <behavior action="append" href="/fragment" target="area3" /> ❸
   Append
  </text>
  <view id="area3">
   <text>Existing content</text>
  </view>

  <text>
   <behavior action="prepend" href="/fragment" target="area4" /> ❹
```

```
   Prepend
  </text>
  <view id="area4">
   <text>Existing content</text>
  </view>

 </body>
</screen>
```

❶ Заменяет элемент area1 полученным фрагментом.
❷ Заменяет дочерние элементы area2 полученным фрагментом.
❶ Присоединяет полученный фрагмент после area3.
❹ Вставляет полученный фрагмент до area4.

В этом примере используется экран с четырьмя кнопками для четырех действий обновления: `replace`, `replace-inner`, `append`, `prepend`. Под каждой кнопкой помещается соответствующий элемент `<view>` с текстом. Обратите внимание: идентификатор каждого `view` отвечает значению `target` поведения соответствующей кнопки.

Когда пользователь нажимает первую кнопку, клиент Hyperview выдает запрос к `/fragment`. Затем он ищет целевой элемент, то есть элемент с идентификатором `area1`. Наконец, он заменяет элемент `<view id="area1">` полученным фрагментом, `<text>My fragment</text>`. Существующий элемент `view` и содержащийся в нем текст будут заменены. Для пользователя все выглядит так, словно текст Existing content («Текущий контент») заменился на My fragment («Мой фрагмент»). В HXML также исчезнет элемент `<view id="area1">`.

Вторая кнопка ведет себя примерно так же, как и первая. Однако действие `replace-inner` не удаляет с экрана целевой элемент, а только заменяет его дочерние элементы. Это означает, что полученная разметка будет иметь вид `<view id="area2"><text>My fragment</text></view>`.

Третья и четвертая кнопки не удаляют никакой контент с экрана. Вместо этого фрагмент будет добавлен либо после дочерних элементов (в случае `append`), либо перед (`prepend`) дочерними элементами целевого элемента.

Для полноты картины рассмотрим состояние экрана после того, как пользователь нажмет все четыре кнопки.

Листинг 179. Действия обновления, экран после нажатия кнопок

```
<screen>
 <body>
  <text>
   <behavior action="replace" href="/fragment" target="area1" />
   Replace
  </text>
```

```
  <text>My fragment</text> ❶

  <text>
   <behavior action="replace-inner" href="/fragment" target="area2" />
   Replace-inner
  </text>
  <view id="area2">
   <text>My fragment</text> ❷
  </view>

  <text>
   <behavior action="append" href="/fragment" target="area3" />
   Append
  </text>
  <view id="area3">
   <text>Existing content</text>
   <text>My fragment</text> ❸
  </view>

  <text>
   <behavior action="prepend" href="/fragment" target="area4" />
   Prepend
  </text>
  <view id="area4">
   <text>My fragment</text> ❹
   <text>Existing content</text>
  </view>

 </body>
</screen>
```

❶ Фрагмент полностью замещает цель действием replace.
❷ Фрагмент заменяет дочерние элементы цели действием replace-inner.
❸ Фрагмент добавляется как последний дочерний элемент действием append.
❹ Фрагмент добавляется как первый дочерний элемент действием prepend.

Приведенные примеры показывают, как действия выдают запросы `GET` на бэкенд. Но эти действия также могут выдавать запросы `POST`, для чего следует задать атрибут `verb="post"` для элемента `<behavior>`. Для запросов `GET` и `POST` данные из родительского элемента `<form>` сериализуются и включаются в запрос. Для запросов `GET` контент кодируется в URL и добавляется в виде параметров запроса. Для запросов `POST` контент кодируется в URL формы и включается в тело запроса. Так как действия обновления поддерживают как `POST`, так и данные форм, они часто используются для отправки данных на бэкенд.

До сих пор в нашем примере действия обновления требовали получения нового контента и добавления его на экран. Но иногда нужно лишь изменить состояние существующих элементов. Наверное, самое распространенное изменение

состояния элемента — переключение его видимости. В Hyperview для этого существуют действия `hide` (скрыть), `show` (показать) и `toggle` (переключить). Как и другие действия обновления, `hide`, `show` и `toggle` используют атрибут `target` для применения к элементу на текущем экране.

Листинг 180. Действия show, hide и toggle

```
<screen>
 <body>
  <text>
   <behavior action="hide" target="area" /> ❶
   Hide
  </text>

  <text>
   <behavior action="show" target="area" /> ❷
   Show
  </text>

  <text>
   <behavior action="toggle" target="area" /> ❸
   Toggle
  </text>

  <view id="area"> ❹
   <text>My fragment</text>
  </view>
 </body>
</screen>
```

❶ Скрывает элемент с идентификатором area.
❷ Отображает элемент с идентификатором area.
❸ Переключает состояние видимости элемента с идентификатором area.
❹ Элемент, являющийся целью действий.

В рассмотренном примере три кнопки, Hide (Скрыть), Show (Показать) и Toggle (Переключить), изменяют состояние видимости элемента `<view>` с идентификатором `area`. Повторное нажатие Hide ничего не изменит, когда представление будет скрыто. Точно так же повторное нажатие Show ни на что не повлияет при отображаемом представлении. Кнопка Toggle переключает состояние видимости элемента между скрытым и отображаемым.

В Hyperview поддерживаются другие действия, изменяющие существующую разметку HXML. Мы не будем рассматривать их подробно, а ограничимся кратким перечислением:

- `set-value` — это действие может задать значение элемента ввода: `<text-field>`, `<switch>`, `<select-single>` и т. д.;

- `select-all` и `unselect-all` работают с элементом `<select-multiple>` для установки/снятия выделения всех вариантов.

Системные действия

Некоторые стандартные действия Hyperview вообще не взаимодействуют с HXML. Вместо этого они открывают доступ к функциональности, предоставляемой мобильной ОС. Например, как Android, так и iOS поддерживают пользовательский интерфейс функции обмена информацией «Share» («Поделиться») системного уровня. Этот пользовательский интерфейс позволяет передавать URL и сообщения из одного приложения в другое. В Hyperview для поддержки этого взаимодействия существует действие `share`. Оно включает нестандартное пространство имен и специфические атрибуты.

Листинг 181. Системное действие share

```
<behavior
 xmlns:share="https://instawork.com/hyperview-share" ❶
 trigger="press"
 action="share" ❷
 share:url="https://www.instawork.com" ❸
 share:message="зацените этот сайт!" ❹
/>
```

❶ Определяет пространство имен для действия share.
❷ Действие этого поведения открывает окно обмена информацией.
❸ Передаваемый URL.
❹ Передаваемое сообщение.

Пространства имен XML нам уже встречались при обсуждении нестандартных элементов. Здесь мы используем пространство имен для атрибутов `url` и `message` элемента `<behavior>`. Атрибутам назначены достаточно общие имена, которые с большой вероятностью могут использоваться другими компонентами и поведениями, так что пространство имен гарантирует отсутствие неоднозначности. При нажатии инициируется действие `share`. Значения атрибутов `url` и `message` будут переданы системному пользовательскому интерфейсу обмена информацией. В нем пользователь сможет поделиться URL и сообщениями через SMS, электронную почту или другие средства коммуникации.

Действие `share` показывает, как действие поведения может использовать нестандартные атрибуты для передачи дополнительных данных, необходимых для взаимодействий. Но некоторые действия требуют еще более структурированных данных, которые могут предоставляться через дочерние элементы `<behavior>`. Hyperview использует эту возможность для реализации действия `alert`. Действие `alert` отображает нестандартное диалоговое окно системного уровня. Для этого диалогового окна необходимо настроить текст заголовка и сообщение,

а также кнопки. Каждая кнопка инициирует другое поведение по нажатию. Такой уровень конфигурации не может быть реализован одними атрибутами, поэтому необходимо использовать нестандартные дочерние элементы для представления поведения каждой кнопки.

Листинг 182. Системное действие alert

```
<behavior
 xmlns:alert="https://hyperview.org/hyperview-alert" ❶
 trigger="press"
 action="alert" ❷
 alert:title="Перейти к следующему экрану, чтобы продолжить?" ❸
 alert:message="Вы уверены, что хотите перейти к следующему экрану?" ❹
>
 <alert:option alert:label="Продолжить"> ❺
  <behavior action="push" href="/next" /> ❻
 </alert:option>
 <alert:option alert:label="Отмена" /> ❼
</behavior>
```

❶ Определяет пространство имен для действия alert.
❷ Действие этого поведения открывает системное диалоговое окно.
❸ Текст заголовка диалогового окна.
❹ Содержимое диалогового окна.
❺ Вариант «Продолжить» в диалоговом окне.
❻ По нажатию «Продолжить» новый экран проталкивается в стек навигации.
❼ Вариант «Отмена» закрывает диалоговое окно.

Как и действие `share`, `alert` использует пространство имен для определения атрибутов и элементов. Сам элемент `<behavior>` содержит атрибуты `title` и `message` диалогового окна. Состав кнопок в диалоговом окне определяется новым элементом `<option>`, вложенным в `<behavior>`. Обратите внимание: каждый элемент `<option>` имеет метку, а также может содержать необязательный элемент `<behavior>`! Эта структура HXML позволяет системному диалоговому окну инициировать любое взаимодействие, которое может определяться как `<behavior>`. В приведенном выше примере нажатие кнопки «Продолжить» открывает новый экран.

Однако с таким же успехом можно инициировать действие обновления для обновления текущего экрана. Также можно открыть интерфейс обмена информацией, или второе диалоговое окно. Но не делайте этого в реальном приложении! С большой силой приходит большая ответственность.

Нестандартные действия

Со стандартными действиями навигации, обновления и системными действиями Hyperview можно построить множество разных мобильных пользовательских

интерфейсов. Однако стандартный набор может не покрывать все взаимодействия, необходимые в мобильном приложении. К счастью, система действий расширяема. По аналогии с тем, как в Hyperview можно добавлять нестандартные элементы, также можно добавлять нестандартные действия поведения. Нестандартные действия имеют синтаксис, сходный с действиями `share` и `alert`, с использованием пространств имен для атрибутов, передающих дополнительные данные. Нестандартные действия также имеют полный доступ к HXML текущего экрана, так что они могут изменять состояние или добавлять/удалять элементы с текущего экрана. В следующей главе мы создадим нестандартное действие поведения, чтобы расширить мобильное приложение для управления контактами.

Триггеры

Ранее мы уже встречали простейший вид триггера — нажатие элемента. Hyperview поддерживает много других стандартных триггеров, используемых в мобильных приложениях.

Долгое нажатие

С нажатием тесно связана операция долгого нажатия. Поведение с `trigger="longPress"` инициируется, когда пользователь выполняет нажатие и удерживает элемент в нажатом состоянии. Взаимодействия «долгого нажатия» часто используются для ускоренного вызова операций и расширенной функциональности. Иногда элементы будут поддерживать разные действия как для нажатия (`press`), так и для долгого нажатия (`longPress`). Для этого используются разные элементы `<behavior>` в одном элементе UI.

Листинг 183. Пример триггера долгого нажатия

```
<text>
 <behavior trigger="press" action="push" href="/next-screen" /> ❶
 <behavior trigger="longPress" action="push" href="/secret-screen" /> ❷
 Нажмите (или нажмите и удерживайте)!
</text>
```

❶ Обычное нажатие открывает следующий экран.

❷ Долгое нажатие открывает другой экран.

В этом примере обычное нажатие открывает новый экран и запрашивает контент от `/next-screen`. Долгое же нажатие открывает новый экран с контентом из `/secret-screen`. Для краткости мы привели надуманный пример. Лучше, если долгое нажатие будет открывать контекстное меню с командами и расширенными функциями. Для этого можно воспользоваться `action="alert"` и открытием системного диалогового окна с кнопками быстрого доступа.

load

Иногда действие должно срабатывать при загрузке экрана. Эту задачу решает `trigger="load"`. Один из возможных сценариев использования — быстрая загрузка «оболочки» экрана с последующим заполнением основного контента экрана вторым действием обновления.

Листинг 184. Пример триггера загрузки

```
<body>
 <view>
  <text>Мое приложение</text>
  <view id="container"> ❶
   <behavior trigger="load" action="replace" href="/content"
target="container"> ❷
   <text>Загрузка...</text> ❸
  </view>
 </view>
</body>
```

❶ Элемент-контейнер без фактического содержимого.

❷ Поведение, которое немедленно запускает запрос к /content для замены контейнера.

❸ Загрузка пользовательского интерфейса, который появляется после загрузки и замены контента.

В этом примере загружается экран с текстом заголовка («Мое приложение»), но без контента. Вместо этого отображается элемент `<view>` с идентификатором `container` и текстом «Загрузка...». Как только экран загрузится, поведение с `trigger="load"` инициирует действие `replace`. Оно запрашивает контент по пути `/content` и заменяет представление-контейнер ответом.

visible

В отличие от `load`, триггер `visible` исполняет поведение только при появлении элемента с поведением в области просмотра на мобильном устройстве в результате прокрутки. Действие `visible` обычно используется для реализации взаимодействия бесконечной прокрутки для списка `<list>` с элементами `<item>`. Последний пункт списка включает поведение с `trigger="visible"`. Действие `append` загружает следующую страницу с пунктами списка и присоединяет их к списку.

refresh

Триггер отражает действие «потянуть, чтобы обновить» для элементов `<list>` и `<view>`. Это взаимодействие связывается с получением обновленного контента от бэкенда. Соответственно, оно обычно выполняется в паре с действием `update` или `reload` для вывода последних данных на экран.

Листинг 185. Пример триггера pull-to-refresh

```
<body>
 <view scroll="true">
  <behavior trigger="refresh" action="reload" /> ❶
  <text>Элементы отсутствуют</text>
 </view>
</body>
```

❶ Когда выполняется действие «потянуть, чтобы обновить», перезагрузить экран.

Обратите внимание: добавление поведения с `trigger="refresh"` в элемент `<view>` или `<list>` добавляет в элемент взаимодействие «потянуть, чтобы обновить», включая отображение спиннера в то время, когда пользователь тянет элемент вниз.

focus, blur и change

Эти триггеры относятся к взаимодействиям с элементами ввода. Соответственно, они инициируют только поведение, присоединенное к таким элементам, как `<text-field>`. `focus` и `blur` срабатывают при получении (focus) и потере фокуса (blur) элементом ввода. `change` срабатывает при изменении значения элемента ввода, например, когда пользователь вводит букву в текстовом поле. Эти триггеры часто используются с поведением, которое должно проверять данные полей формы на стороне сервера. Например, когда пользователь вводит имя пользователя, а затем передает фокус другому полю, поведение может срабатывать по `blur` для выдачи запроса на бэкенд и проверки уникальности имени. Если введенное имя неуникально, ответ может включать сообщение об ошибке, которое предлагает пользователю выбрать другое имя.

Множественное поведение

Во многих примерах, приведенных выше, к элементу присоединяется один элемент `<behavior>`. Однако в Hyperview таких ограничений нет; элементы могут определять несколько вариантов поведения. Вы уже видели пример, в котором одному элементу назначалось несколько действий, срабатывавших по `press` и `longPress`. Также по одному триггеру могут срабатывать сразу несколько действий.

В этом (откровенно говоря, надуманном) примере при нажатии кнопки Hide (Скрыть) на экране должны скрываться два элемента. Два элемента расположены достаточно далеко друг от друга в HXML, и их не удастся скрыть, скрывая общий родительский элемент. Но можно инициировать два поведения одновременно, каждое из которых будет выполнять действие `hide` для своего элемента.

Листинг 186. Варианты множественного поведения, срабатывающие по нажатию

```
<screen>
 <body>
  <text id="area1">Область 1</text>

  <text>
   <behavior trigger="press" action="hide" target="area1" /> ❶
   <behavior trigger="press" action="hide" target="area2" /> ❷
   Скрыть
  </text>

  <text id="area2"> Область 2</text>
 </body>
</screen>
```

❶ По нажатию скрывает элемент с идентификатором area1.
❷ По нажатию скрывает элемент с идентификатором area2.

Hyperview обрабатывает варианты поведения в порядке их следования в разметке. В данном случае сначала будет скрыт элемент с идентификатором `area1`, а потом элемент с идентификатором `area2`. Так как `hide` является мгновенным действием (то есть не выдает запрос HTTP), для пользователя оба элемента будут исчезать одновременно. Но что, если бы мы инициировали два действия, зависящие от ответов на запросы HTTP (как с `replace-inner`)? В таком случае каждое отдельное действие обрабатывается сразу же при получении Hyperview ответа HTTP. В зависимости от сетевой задержки два действия могут применяться в любом порядке, и их одновременное применение не гарантировано.

Мы рассмотрели элементы с несколькими вариантами поведения и разными триггерами. И вы уже видели элементы с несколькими вариантами поведения с одним триггером. Эти концепции также могут смешиваться. Реальное приложение Hyperview нередко содержит несколько вариантов поведения, часть из которых инициируется вместе, а другие — по разным взаимодействиям. Использование нескольких вариантов поведения с несколькими действиями поддерживает декларативность HXML без ущерба для функциональности.

Итоги

В этой главе мы рассмотрели ряд новых концепций, и это было очень краткое введение в HXML. Если вы захотите узнать об HXML больше, мы рекомендуем обратиться к официальной справочной документации[1]. А пока надеемся, что вам хватит нескольких ключевых выводов.

[1] https://hyperview.org/docs/reference_index

Во-первых, HXML похож на HTML. Веб-разработчики, хорошо знакомые с фреймворками рендеринга на стороне сервера, могут использовать эти методы для написания HXML. Кроме базовых элементов UI (`<view>`, `<text>`, `<image>`), HXML определяет элементы для реализации пользовательского интерфейса, специфичного для мобильных устройств. К этой категории относятся как паттерны макетов (`<screen>`, `<list>`, `<section-list>`), так и элементы ввода (`<switch>`, `<select-single>`, `<select-multiple>`).

Во-вторых, взаимодействия в HXML определяются при помощи поведения. Элементы `<behavior>`, действуя по образцу `htmx`, разделяют пользовательские взаимодействия (триггеры) и результирующие действия. Действия поведений делятся на три широкие категории:

- действия навигации (`push`, `back`) обеспечивают перемещения между экранами мобильного приложения;
- действия обновления (`replace`, `append`) обновляют экран новыми фрагментами HXML, запрашиваемыми с сервера;
- системные действия (`alert`, `share`) обеспечивают взаимодействие с функциональностью системного уровня в iOS и Android.

Наконец, сам язык HXML проектировался для настраиваемых элементов. Разработчики могут задавать нестандартные элементы и нестандартные действия поведения, чтобы расширять набор возможных взаимодействий пользователей с создаваемыми приложениями.

Гипермедиа для мобильных устройств

Разработка гипермедиа-управляемых приложений на мобильных устройствах вполне обоснованна. Платформы мобильных приложений подталкивают разработчиков к применению архитектуры толстого клиента. Однако приложения, использующие толстые клиенты, сталкиваются с теми же проблемами, что и приложения SPA в веб-среде. Мобильная архитектура гипермедиа способна решить эти проблемы.

Технология Hyperview, построенная на основе нового формата HXML, обеспечивает эту возможность. Она предоставляет мобильный тонкий клиент с открытым исходным кодом для рендеринга HXML. Со своей стороны HXML предоставляет инструментарий элементов и паттернов для мобильных UI. Разработчики могут модифицировать Hyperview под требования своих приложений, при этом полностью сохраняя архитектуру гипермедиа, — это несомненный плюс.

Да, гипермедиа может работать и в мобильных приложениях. В следующих двух главах мы покажем, как преобразовать веб-приложение Contact.app в нативное мобильное приложение на основе Hyperview.

Заметки о гипермедиа: максимизируйте преимущества серверного кода

Так как в разделах книги, посвященных Hyperview, мы не используем HTML, основное внимание мы уделим гипермедиа, а не соображениям и советам, касающимся HTML.

Большое преимущество гипермедиа-управляемого подхода заключается в том, что с ним окружение на стороне сервера начинает играть намного более важную роль при построении веб-приложений. Вместо того чтобы просто производить JSON, бэкенд становится неотъемлемым компонентом взаимодействия пользователя с приложением гипермедиа.

По этой причине стоит лучше разобраться в доступной функциональности. Например, во многих старых веб-фреймворках очень хорошо развита функциональность, связанная с генерированием HTML. Такая функциональность, как кэширование на стороне сервера, может стать решающим фактором, определяющим каким будет приложение — динамичным, с быстрым откликом, или же заторможенным.

Не жалейте времени на изучение всех доступных инструментов.

Хорошее практическое правило: на получение ответа сервера в гипермедиа-управляемом приложении должно уходить менее 100 миллисекунд, а проверенные фреймворки на стороне сервера предоставляют средства, которые помогут достичь этого показателя.

В средах на стороне сервера часто существуют отлично проработанные механизмы для правильной факторизации (или организации) кода. Паттерн «модель — представление — контроллер» хорошо развит в большинстве сред, и такие средства, как модули, пакеты и т. д., предоставляют отличный способ организации кода.

Современные приложения SPA и мобильные пользовательские интерфейсы обычно строятся из компонентов, а гипермедиа-управляемые приложения — на добавлении шаблонов, где шаблоны на стороне сервера делятся в соответствии с потребностями приложения в рендеринге гипермедиа, а затем добавляются друг в друга по мере надобности. Обычно такой подход приводит к тому, что файлов становится меньше, чем в типичном компонентном приложении, и они увеличиваются в размерах.

Другая технология, на которую стоит обратить внимание, — фрагменты шаблонов, позволяющие отрендерить только часть файла шаблона. Она может еще сократить количество файлов шаблонов, необходимых для приложения на стороне сервера.

Также мы рекомендуем воспользоваться прямым доступом к хранилищу данных. Когда приложение строится по схеме толстого клиента, хранилище данных обычно находится за API данных (например, JSON). Дополнительный уровень абстракции часто мешает фронтенд-разработчикам в полной мере пользоваться средствами, доступными в хранилище данных. Здесь может помочь, например, GraphQL, но использование этой среды создает проблемы безопасности, недостаточно хорошо понимаемые многими разработчиками.

Вместе с тем, когда информация гипермедиа генерируется на стороне сервера, разработчик получает неограниченный доступ к хранилищу данных и может пользоваться, например, функциями соединения и агрегирования в хранилищах SQL.

При этом в распоряжении разработчика, производящего финальные данные гипермедиа, появляются намного более выразительные инструменты. Так как API гипермедиа может структурироваться вокруг потребностей пользовательского интерфейса, можно настроить каждую точку данных так, чтобы она выдавала как можно меньше запросов к хранилищу.

Практическое правило: каждый запрос к серверу должен выдавать не более трех обращений к хранилищу данных. Если это правило соблюдать, у гипермедиа-управляемого приложения не будет никаких проблем со временем отклика.

СОЗДАНИЕ ПРИЛОЖЕНИЯ КОНТАКТОВ С ИСПОЛЬЗОВАНИЕМ HYPERVIEW

В предыдущих главах книги объяснялись преимущества построения приложений с использованием архитектуры гипермедиа. Для демонстрации этих преимуществ мы создали веб-приложение Contacts. Затем в главе 11 вы узнали, что концепции гипермедиа могут (и должны) применяться не только в веб-среде, но и на других платформах. В частности, мы представили Hyperview как пример формата гипермедиа и клиента, разработанного специально для построения мобильных приложений. Но, возможно, у читателей остался вопрос: как происходит создание полнофункциональных мобильных приложений с использованием Hyperview? Придется ли изучать совершенно новый язык и фреймворк? В этой главе мы представим Hyperview в действии, для чего портируем веб-приложение Contacts в нативное мобильное приложение. Вы увидите, что многие методы веб-разработки (и на самом деле бóльшая часть кода) вообще не изменяются при разработке с Hyperview. Как такое возможно?

1. Наше веб-приложение Contacts строилось по принципу HATEOAS (Hypermedia As The Engine of Application State, «гипермедиа как ядро состояния приложения»). Вся функциональность приложения (загрузка, поиск, редактирование и создание контактов) реализуется в бэкенде (класс Python `Contacts`). Наше мобильное приложение, построенное на основе Hyperview, тоже использует HATEOAS и размещает всю логику в бэкенде. Это означает, что класс Python `Contacts` может обеспечивать работу мобильного приложения точно так же, как он обеспечивает работу веб-приложения, — никаких изменений.
2. Коммуникации «клиент — сервер» в веб-приложении происходят по протоколу HTTP. Сервер HTTP для веб-приложения написан с использованием фреймворка Flask. Hyperview также использует HTTP для коммуникаций «клиент — сервер». Следовательно, мы сможем использовать маршруты Flask и представления из веб-приложения и в мобильном приложении.

3. В качестве фомата гипермедиа веб-приложение использует HTML, а Hyperview — HXML. HTML и HXML — разные форматы, но их базовый синтаксис похож (вложенные теги и атрибуты). Это означает, что мы можем использовать одну библиотеку шаблонов (Jinja) для HTML и HXML. Кроме того, многие концепции `htmx` встроены в HXML. Вы сможете напрямую портировать функциональность веб-приложения, реализованную средствами `htmx` (поиск, бесконечная загрузка), в HXML.

В сущности, можно повторно использовать практически весь код бэкенда веб-приложения, но шаблоны HTML нужно будет заменить шаблонами HXML. Во многих разделах этой главы предполагается, что веб-приложение для управления контактами выполняется локально и ведет прослушивание через порт 5000.

Готовы? Создадим несколько новых шаблонов HXML для пользовательского интерфейса нашего мобильного приложения.

Создание мобильного приложения

Чтобы приступить к работе с HXML, вам понадобится клиент Hyperview. При разработке веб-приложений придется позаботиться только о сервере, поскольку клиент (веб-браузер) доступен повсеместно. Однако не существует эквивалентного клиента Hyperview, установленного на каждом мобильном устройстве. Вместо него мы создаем собственный клиент Hyperview, настроенный на взаимодействие только с нашим сервером. Этот клиент можно упаковать в мобильное приложение Android или iOS и распространять через соответствующие магазины приложений.

К счастью, реализацию клиента Hyperview не нужно начинать с нуля. Репозиторий кода Hyperview содержит демоверсию бэкенда и демоверсию клиента, построенную с использованием Expo. Мы воспользуемся этим клиентом, но в качестве отправной точки укажем бэкенд приложения.

```
> git clone git@github.com:Instawork/hyperview.git
> cd hyperview/demo
> yarn ❶
> yarn start ❷
```

❶ Устанавливает зависимости для демоверсии приложения.

❷ Запускает сервер Expo для запуска мобильного приложения в симуляторе iOS.

После выполнения команды `yarn start` открывается приглашение, в котором предлагается запустить мобильное приложение в эмуляторе Android или симуляторе iOS. Выберите нужный вариант в зависимости от установленной версии SDK разработчика. (Снимки экранов в этой главе сделаны в симуляторе iOS.) Если все нормально, в симуляторе должно быть установлено мобильное при-

ложение Expo. Мобильное приложение автоматически запускается и выводит экран с сообщением о неудачном сетевом запросе (Network request failed). Дело в том, что приложение по умолчанию настроено на выдачу запроса по адресу http://0.0.0.0:8085/index.xml, а бэкенд нашего приложения прослушивает порт 5000. Для решения этой проблемы можно внести простое изменение в конфигурацию в файле demo/src/constants.js:

```
//export const ENTRY_POINT_URL = 'http://0.0.0.0:8085/index.xml'; ❶
export const ENTRY_POINT_URL = 'http://0.0.0.0:5000/'; ❷
```

❶ URL точки входа по умолчанию в демоверсии.

❷ Заменяется URL приложения для управления контактами.

Впрочем, это еще не все. Теперь, когда клиент Hyperview указывает на правильную конечную точку, выводится другая ошибка: `ParseError`. Это связано с тем, что бэкенд отвечает на запросы контентом HTML, но клиент Hyperview ожидает получить ответ XML (а конкретно HXML). Пришло время заняться бэкендом на основе Flask. Мы рассмотрим представления Flask и заменим шаблоны HTML шаблонами HXML. А конкретно, в нашем мобильном приложении будут поддерживаться следующие возможности:

- список контактов с поддержкой поиска;
- просмотр подробной информации контакта;
- редактирование контакта;
- удаление контакта;
- добавление нового контакта.

НУЛЕВАЯ КОНФИГУРАЦИЯ НА СТОРОНЕ КЛИЕНТА В ПРИЛОЖЕНИЯХ ГИПЕРМЕДИА

Если мобильное приложение использует клиент Hyperview, часто единственным клиентским кодом, который придется написать, чтобы оно было полнофункциональным, становится код настройки URL точки входа. Считайте URL точки входа адресом, который вводится в веб-браузере для открытия веб-приложения. Правда, в Hyperview нет адресной строки и браузер жестко запрограммирован для открытия только одного URL. По этому URL загружается первый экран при запуске приложения. Любое другое действие, которое выполняется пользователем, будет объявляться в разметке HXML первого экрана. Минимальная конфигурация – одно из преимуществ гипермедиа-управляемой архитектуры.

Конечно, вы захотите написать больше кода на устройстве, чтобы ваше приложение поддерживало больше функциональности. Мы покажем, как это делается, далее в этой главе (раздел «Расширение клиента Hyperview»).

Список контактов с поддержкой поиска

Построение приложения Hyperview начинается с экрана точки входа — списка контактов. В исходной версии этого экрана будет поддерживаться следующая функциональность веб-приложения.

- Вывод списка контактов с поддержкой прокрутки.
- Поле с «живым поиском» («search-as-you-type») над списком.
- «Бесконечная прокрутка» для загрузки новых контактов по мере прокрутки списка пользователем.

Кроме того, мы добавим взаимодействие «потянуть, чтобы обновить» для списка, так как пользователи ожидают такого поведения в списковых интерфейсах мобильных приложений.

Напомним, что все страницы веб-приложения Contacts расширяют общий базовый шаблон layout.html. Нам понадобится аналогичный базовый шаблон для экранов мобильного приложения. Базовый шаблон содержит правила стилей пользовательского интерфейса и основную структуру, общую для всех экранов. Назовем файл шаблона layout.xml.

Листинг 187. Базовый шаблон hv/layout.xml

```
<doc xmlns="https://hyperview.org/hyperview">
 <screen>
  <styles><!-- Фрагмент опущен для краткости --></styles>
  <body style="body" safe-area="true">
   <header style="header">
    {% block header %} ❶
     <text style="header-title">Contact.app</text>
    {% endblock %}
   </header>

   <view style="main">
    {% block content %}{% endblock %} ❷
   </view>
  </body>
 </screen>
</doc>
```

❶ Заголовок в шаблоне с текстом по умолчанию.

❷ Секция контента в шаблоне, заполняемая другими шаблонами.

В разметке используются теги HXML и атрибуты, рассмотренные в предыдущей главе. Шаблон создает базовый макет экрана с тегами `<doc>`, `<screen>`, `<body>`, `<header>` и `<view>`. Заметим, что синтаксис HXML хорошо сочетается с библиотекой шаблонов Jinja. Здесь мы используем блоки Jinja для определения двух секций (заголовок и контент), которые совместно образуют уникальный контент

экрана. Когда базовый шаблон будет готов, можно переходить к созданию шаблона для экрана со списком контактов.

Листинг 188. Начало файла hv/index.xml

```
{% extends 'hv/layout.xml' %} ❶

{% block content %} ❷
 <form> ❸
  <text-field name="q" value="" placeholder="Search..." style="search-
field" />
  <list id="contacts-list"> ❹
   {% include 'hv/rows.xml' %}
  </list>
 </form>
{% endblock %}
```

❶ Расширяет базовый шаблон layout.
❷ Переопределяет блок контента шаблона layout.
❸ Создает форму поиска, которая выдает запрос HTTP GET к /contacts.
❹ Список контактов использует тег include Jinja.

Шаблон расширяет базовый шаблон layout.xml и переопределяет блок контента элементом `<form>`. На первый взгляд может показаться странным, что форма включает и `<text-field>`, и `<list>`. Но вспомните: в Hyperview данные формы включаются в любые запросы, исходящие от дочернего элемента. Вскоре мы добавим к списку взаимодействия («потянуть, чтобы обновить»), требующие данных формы. Обратите внимание на использование тега Jinja `include` для рендеринга HXML для строк контактов в списке (hv/rows.xml). Как и в шаблонах HTML, `include` может использоваться для разбиения HXML на меньшие части. Тег также позволяет серверу отвечать только шаблоном rows.xml на такие взаимодействия, как поиск, бесконечная прокрутка и обновление.

Листинг 189. hv/rows.xml

```
<items xmlns="https://hyperview.org/hyperview"> ❶
 {% for contact in contacts %} ❷
  <item key="{{ contact.id }}" style="contact-item"> ❸
   <text style="contact-item-label">
    {% if contact.first %}
     {{ contact.first }} {{ contact.last }}
    {% elif contact.phone %}
     {{ contact.phone }}
    {% elif contact.email %}
     {{ contact.email }}
    {% endif %}
   </text>
  </item>
```

```
 {% endfor %}
</items>
```

❶ Элемент HXML, группирующий набор элементов <item> в общем родителе.
❷ Перебирает контакты, передаваемые шаблону.
❸ Рендерит <item> для каждого контакта с выводом имени, номера телефона или адреса электронной почты.

В веб-приложении каждая строка списка содержит имя контакта, номер телефона и адрес электронной почты. Но в мобильном приложении меньше свободного места, и трудно втиснуть всю эту информацию в одну строку. Поэтому в строке выводятся только имя и фамилия контакта, а если имя не задано, приложение показывает адрес электронной почты или телефон. Чтобы отрендерить строку, мы снова используем синтаксис шаблона Jinja для рендеринга динамического текста с данными, переданными шаблону.

Теперь у нас есть шаблоны для базового макета, экрана контактов и строк контактов. При этом еще необходимо обновить представления Flask, чтобы они использовали шаблон. Посмотрите, как выглядит текущее представление `contacts()` для веб-приложения.

Листинг 190. app.py

```
@app.route("/contacts")
def contacts():
  search = request.args.get("q")
  page = int(request.args.get("page", 1))
  if search:
    contacts_set = Contact.search(search)
    if request.headers.get('HX-Trigger') == 'search':
      return render_template("rows.html", contacts=contacts_set,
page=page)
  else:
    contacts_set = Contact.all(page)
  return render_template("index.html", contacts=contacts_set, page=page)
```

Представление поддерживает выборку группы контактов на основании двух параметров запроса, `q` и `page`. Оно также решает, следует ли рендерить полную страницу (index.html) или только строки контактов (rows.html) в зависимости от заголовка `HX-Trigger`. Здесь возникает небольшая проблема. Заголовок `HX-Trigger` устанавливается библиотекой `htmx`; в Hyperview подобная функциональность отсутствует. Более того, в следующих сценариях ответ должен содержать только строки контактов:

- поиск;
- pull-to-refresh («потянуть, чтобы обновить»);
- загрузка следующей страницы контактов.

Так как мы не можем зависеть от заголовка (такого, как `HX-Trigger`), понадобится другой способ определить, нужен ли клиенту в ответе полный экран или только строки. Для этого мы введем новый параметр запроса, `rows_only`. Если значение этого параметра равно `true`, то представление отвечает на запрос рендерингом шаблона rows.xml. В противном случае оно рендерит шаблон index.xml.

Листинг 191. app.py

```
@app.route("/contacts")
def contacts():
  search = request.args.get("q")
  page = int(request.args.get("page", 1))
  rows_only = request.args.get("rows_only") == "true" ❶
  if search:
    contacts_set = Contact.search(search)
  else:
    contacts_set = Contact.all(page)

  template_name = "hv/rows.xml" if rows_only else "hv/index.xml" ❷
  return render_template(template_name, contacts=contacts_set, page=page)
```

❶ Проверяет новый параметр rows_only query.

❷ Рендерит шаблон HXML в зависимости от значения rows_only.

Остается внести еще одно изменение. Flask предполагает, что большинство представлений в ответе использует HTML, поэтому по умолчанию заголовок ответа `Content-Type` содержит значение `text/html`. Но клиент Hyperview ожидает получить контент HXML, на что указывает заголовок ответа `Content-Type` со значением `application/vnd.hyperview+xml`. Клиент не принимает ответы с другим типом контента. Чтобы исправить несоответствие, необходимо явно задать заголовок ответа `Content-Type` в представлениях Flask. Для этого добавим в приложение новую функцию-хелпер `render_to_response()`.

Листинг 192. app.py

```
def render_to_response(template_name, *args, **kwargs):
  content = render_template(template_name, *args, **kwargs) ❶
  response = make_response(content) ❷
  response.headers['Content-Type'] = 'application/vnd.hyperview+xml' ❸
  return response
```

❶ Рендерит шаблон с заданными аргументами и ключевыми аргументами.

❷ Создает объект response с готовым шаблоном.

❸ Присваивает заголовку ответа Content-Type значение XML.

Как видно из листинга, функция-хелпер использует во внутренней реализации `render_template()`. `render_template()` возвращает строку. С помощью этой

строки хелпер создает объект `Response`. Объект содержит атрибут `headers`, позволяющий задавать и изменять заголовки ответа. А конкретно `render_to_response()` присваивает `Content-Type` значение `application/vnd.hyperview+xml`, чтобы клиент Hyperview распознал контент. Функция-хелпер напрямую заменяет `render_template` в представлениях. Таким образом, все, что потребуется, — обновить последнюю строку функции `contacts()`.

Листинг 193. Функция contacts()

```
return render_to_response(template_name, contacts=contacts_set, page=page) ❶
```

❶ Рендерит шаблон HXML в ответ XML.

После всех изменений в представлении `contacts()` мы наконец-то видим плоды своего труда на экране. После перезапуска бэкенда и обновления экрана в мобильном приложении открывается список контактов!

Рис. 13. Экран со списком контактов

Поиск контактов

К настоящему моменту мы имеем мобильное приложение, которое выводит на экран список контактов. Однако пользовательский интерфейс пока не поддерживает никаких взаимодействий. При вводе запроса в поле поиска список контактов не фильтруется. Добавим в поле поиска поведение, реализующее «живой поиск», то есть поиск в процессе ввода текста («search-as-you-type»). Для этого следует расширить `<text-field>` и добавить элемент `<behavior>`.

Листинг 194. Фрагмент hv/index.xml

```
<text-field name="q" value="" placeholder="Search..." style="search-field">
 <behavior
  trigger="change" ❶
  action="replace-inner" ❷
  target="contacts-list" ❸
  href="/contacts?rows_only=true" ❹
  verb="get" ❺
 />
</text-field>
```

❶ Поведение срабатывает при изменении значения текстового поля.
❷ При срабатывании поведения действие заменяет контент внутри целевого элемента.
❸ Целевым для действия является элемент с идентификатором contacts-list.
❹ Заменяющий контент загружается по этому URL.
❺ Для загрузки заменяющего контента используется метод HTTP GET.

Первое, что можно заметить: текстовое поле из самозакрывающегося тега (`<text-field />`) было преобразовано в пару из открывающего и закрывающего тегов (`<text-field>... </text-field>`). Это позволяет добавить дочерний элемент `<behavior>` для определения взаимодействия.

Атрибут `trigger="change"` сообщает Hyperview, что изменение в значении текстового поля инициирует действие. Каждый раз, когда пользователь редактирует содержимое текстового поля, добавляя или удаляя символы, инициируется действие.

Остальные атрибуты элемента `<behavior>` определяют действие. Атрибут `action="replace-inner"` означает, что действие обновляет содержимое экрана, заменяя контент HXML элемента новым контентом. Чтобы операция `replace-inner` сделала то, что положено, необходимо знать две вещи: текущий элемент на экране, который станет целевым для действия, и контент, который будет использован для замены. `target="contacts-list"` сообщает идентификатор текущего элемента. Значение `id="contacts-list"` присваивается в элементе `<list>` в index.xml. Таким образом, когда пользователь вводит запрос в текстовом

поле, Hyperview заменяет содержимое `<list>` (группа элементов `<item>`) новым контентом (элементы `<item>`, соответствующие критерию запроса), полученным в относительном `href` из ответа. Домен определяется по домену, использованному для загрузки экрана. Обратите внимание: `href` включает только параметр `rows_only`; то есть нам нужно, чтобы ответ включал только строки, а не весь экран.

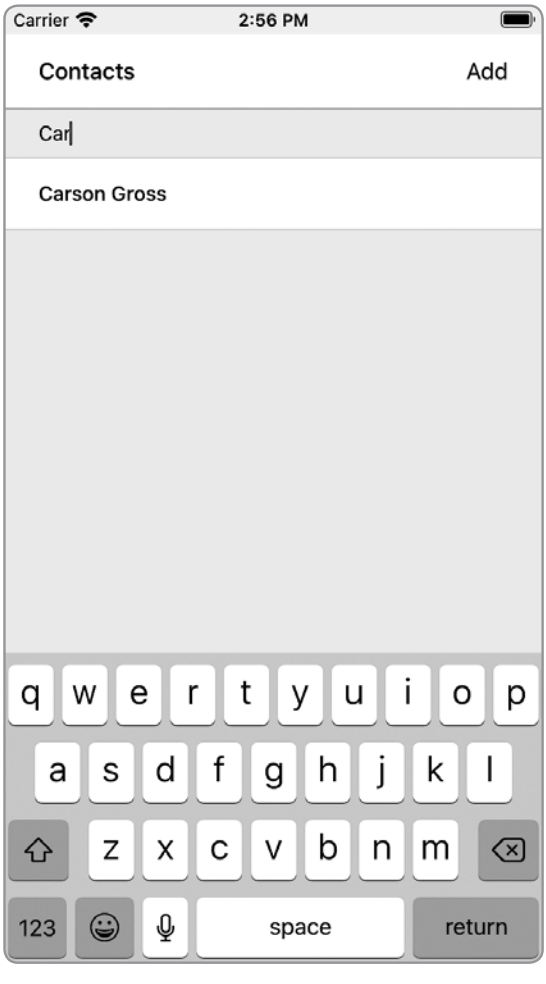


Рис. 14. Поиск контакта при вводе

И это все, что требуется для добавления в мобильное приложение функциональности поиска при вводе! По мере ввода пользователем поискового запро-

са клиент выдает запросы на бэкенд и заменяет список результатами поиска. Возможно, вас интересует, как бэкенд определяет, какой запрос использовать? Атрибут `href` в поведении не включает параметр `q`, который ожидает получить бэкенд. Но помните, что в index.xml мы заключили элементы `<text-field>` и `<list>` в родительский элемент `<form>`. Элемент `<form>` определяет группу полей ввода, которые сериализуются и включаются в любые запросы HTTP, инициируемые его дочерними элементами. В данном случае элемент `<form>` заключает поведение поиска и текстовое поле. Таким образом, значение `<text-field>` будет включено в запрос HTTP для результатов поиска. Так как выдается запрос `GET`, имя и значение текстового поля будут сериализованы как параметр запроса. Все существующие параметры запроса в `href` будут сохранены. Это означает, что фактический запрос HTTP на бэкенд имеет вид `GET /contacts?rows_only=true&q=Car`. Бэкенд уже поддерживает параметр `q` для поиска, так что ответ будет включать строки данных, соответствующие тексту «Car».

Бесконечная прокрутка

Если пользователь работает с сотнями и тысячами контактов, их одновременная загрузка может привести к снижению производительности приложения. По этой причине во многих мобильных приложениях с длинными списками реализуется взаимодействие, известное как «бесконечная прокрутка». Приложение загружает фиксированное количество начальных элементов списка — допустим, 100. Когда пользователь прокручивает список до конца, он видит спиннер, означающий, что загружается дополнительный контент. Когда контент станет доступен, спиннер заменяется следующей страницей из 100 элементов списка. Новое содержимое присоединяется к списку, не заменяя первую группу элементов. Теперь список содержит 200 элементов. Если пользователь снова прокрутит список до конца, он увидит новый спиннер, а приложение загрузит следующую часть контента. Бесконечная прокрутка улучшает производительность приложения по нескольким параметрам:

- исходный запрос 100 элементов списка будет обработан быстро с предсказуемой задержкой;
- последующие запросы также будут обработаны быстро и предсказуемо;
- если пользователь не прокрутил список до конца, приложение не будет выдавать последующие запросы.

Бэкенд Flask уже поддерживает разбивку на страницы по конечной точке `/contacts` через параметр запроса page. Чтобы воспользоваться этим параметром, необходимо изменить шаблоны HXML. Отредактируем файл rows.xml и добавим новый элемент `<item>` под `for`-циклом Jinja.

Листинг 195. Фрагмент hv/rows.xml

```
<items xmlns="https://hyperview.org/hyperview">
 {% for contact in contacts %}
  <item key="{{ contact.id }}" style="contact-item">
   <!-- Фрагмент опущен для краткости -->
  </item>
 {% endfor %}
 {% if contacts|length > 0 %}
  <item key="load-more" id="load-more" style="load-more-item"> ❶
   <behavior
    trigger="visible" ❷
    action="replace" ❸
    target="load-more" ❹
    href="/contacts?rows_only=true&page={{ page + 1 }}" ❺
    verb="get"
   />
   <spinner /> ❻
  </item>
 {% endif %}
</items>
```

❶ Включает в список дополнительный элемент <item> для отображения спиннера.
❷ Поведение элемента инициируется, когда он становится видимым в области просмотра.
❸ Запущенное поведение заменяет элемент на экране.
❹ Заменяется сам элемент (идентификатор load-more).
❺ Элемент заменяется следующей страницей контента.
❻ Элемент-спиннер.

Если переданный шаблону текущий список контактов пуст, можно предположить, что контактов для получения от бэкенда не осталось. Соответственно, мы используем условную конструкцию Jinja, чтобы новый элемент `<item>` включался только в том случае, если список контактов не пуст. Новому элементу `<item>` назначаются идентификатор и поведение. Поведение определяет взаимодействие бесконечной прокрутки.

До сих пор в коде встречались триггеры `change` и `refresh`. Но чтобы реализовать бесконечную прокрутку, необходимо инициировать действие, когда пользователь прокручивает список до конца. Для этого можно воспользоваться триггером `visible`. Он инициирует действие, когда элемент с поведением становится видимым в области просмотра устройства. В данном случае новый элемент `<item>` является последним элементом списка, так что действие сработает, когда пользователь прокрутит список достаточно далеко, чтобы элемент вошел в область просмотра. Как только элемент становится видимым, действие выдает запрос HTTP `GET` и заменяет загружающийся элемент `<item>` контентом ответа.

Обратите внимание: ссылка должна содержать параметр запроса `rows_only=true`, чтобы наш запрос включал HXML только для элементов контактов,

а не для всего экрана. Кроме того, передается параметр запроса `page` с увеличенным номером текущей страницы, чтобы обеспечить загрузку следующей страницы.

Что произойдет, если контакты занимают более одной страницы? Исходный экран включает первые 100 элементов, а также элемент `load-more` в нижней части. Когда пользователь прокручивает список до конца, Hyperview запрашивает вторую страницу (`&page=2`) и заменяет элемент `load-more` новыми элементами. Но вторая страница будет включать новый элемент `load-more`. Таким образом, когда пользователь прокрутит все элементы второй страницы, Hyperview снова запросит новую порцию элементов (`&page=3`). И снова элемент `load-more` будет замещен новыми элементами. Процедура продолжится, пока все элементы не будут загружены на экран. Тогда контактов не останется, ответ не будет включать очередной элемент `load-more` и разбивка на страницы будет завершена.

Pull-to-refresh

Pull-to-refresh («потянуть, чтобы обновить») — стандартное взаимодействие в мобильных приложениях, особенно на экранах с динамическим контентом. Оно работает следующим образом: в верхней части представления с прокруткой пользователь тянет прокручиваемый контент вниз жестом смахивания. Под контентом появляется спиннер. Если контент был смещен достаточно далеко вниз, инициируется обновление. Пока контент обновляется, спиннер остается на экране, показывая пользователю, что действие все еще выполняется. Когда обновление завершается, контент возвращается в свою стандартную позицию, а спиннер скрывается. Это сообщает пользователю о том, что взаимодействие завершено.

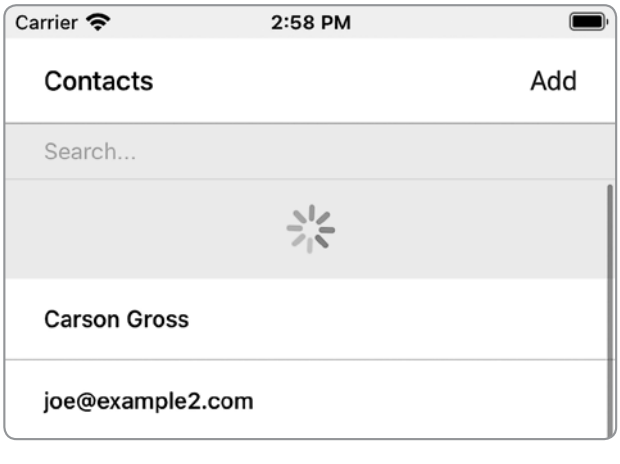


Рис. 15. Действие pull-to-refresh

Этот паттерн настолько широко распространен и полезен, что он встроен в Hyperview в виде действия `refresh`. Добавим это взаимодействие в список контактов, чтобы увидеть, как он работает.

Листинг 196. Фрагмент hv/index.xml

```
<list id="contacts-list"
 trigger="refresh" ❶
 action="replace-inner" ❷
 target="contacts-list" ❸
 href="/contacts?rows_only=true" ❹
 verb="get" ❺
>
 {% include 'hv/rows.xml' %}
</list>
```

❶ Поведение инициируется, когда пользователь выполняет жест pull-to-refresh .
❷ При инициировании поведения действие заменяет контент внутри целевого элемента.
❸ Целевым для действия является сам элемент <list>.
❹ Заменяющий контент загружается по этому URL.
❺ Для загрузки заменяющего контента используется метод HTTP GET.

В приведенном фрагменте можно заметить кое-что необычное: вместо того чтобы добавлять элемент `<behavior>` в `<list>`, мы добавили атрибуты поведения прямо в элемент `<list>`. Это сокращенная запись, иногда удобная для определения одиночных поведений для элемента. Она эквивалентна добавлению в `<list>` элемента `<behavior>` с теми же атрибутами.

Почему же мы воспользовались здесь сокращенной записью? Ответ связан с действием `replace-inner`. Напомним, что это действие заменяет все дочерние элементы целевого элемента новым контентом. Но в эту категорию также входят элементы `<behavior>`! Допустим, наш список `<list>` содержал `<behavior>`. Если пользователь провел поиск или обновил список жестом, содержимое `<list>` заменяется содержимым из rows.xml. Элемент `<behavior>` уже не будет определен в `<list>`, и следующие попытки провести пальцем по экрану, чтобы обновить список, не будут работать. Когда мы определяем поведение в виде атрибутов `<list>`, оно сохраняется даже при замене контента в списке. В общем случае мы предпочитаем явно задавать элементы `<behavior>` в HXML — это упрощает определение нескольких поведений, а также перемещение поведения в процессе рефакторинга. Однако сокращенный синтаксис удобен в ситуациях, подобных описанной выше.

Просмотр подробной информации о контакте

Итак, мы привели в порядок экран со списком контактов, и теперь можно добавлять в приложение другие экраны. Следующим логичным шагом будет

создание экрана подробной информации, который отображается, когда пользователь касается одного из пунктов в списке контактов. Обновим шаблон, который рендерит элементы `<item>` с данными контактов, и добавим поведение для вывода экрана с подробной информацией.

Листинг 197. hv/rows.xml

```
<items xmlns="https://hyperview.org/hyperview">
 {% for contact in contacts %}
  <item key="{{ contact.id }}" style="contact-item">
   <behavior trigger="press" action="push" href="/contacts/{{ contact.id
}}" /> ❶
   <text style="contact-item-label">
    <!—Фрагмент опущен для краткости -->
   </text>
  </item>
 {% endfor %}
</items>
```

❶ Поведение для проталкивания экрана с подробной информацией о контакте в стек по нажатию на контакте.

Бэкенд Flask уже имеет маршрут для предоставления подробной информации о контакте по адресу `/contacts/<contact_id>`. В нашем шаблоне переменная Jinja используется для динамического генерирования пути URL для текущего контакта в цикле `for`. Мы также применили действие `push` для вывода подробной информации с проталкиванием нового экрана в стек. Если перезагрузить приложение, при касании любого контакта в списке Hyperview будет открывать новый экран. Однако на нем будет выводиться сообщение об ошибке.

Дело в том, что бэкенд все еще возвращает HTML в ответе, а клиент Hyperview ожидает получить HXML. Обновим бэкенд, чтобы он отвечал HXML с верными заголовками.

Листинг 198. app.py

```
@app.route("/contacts/<contact_id>")
def contacts_view(contact_id=0):
  contact = Contact.find(contact_id)
  return render_to_response("hv/show.xml", contact=contact) ❶
```

❶ Генерирует ответ XML по новому файлу шаблона.

Как и представление `contacts()`, `contacts_view()` использует `render_to_response()` для назначения заголовка `Content-Type` в ответе. Мы также генерируем ответ по новому шаблону HXML, который создадим сейчас.

Листинг 199. hv/show.xml

```
{% extends 'hv/layout.xml' %} ❶

{% block header %} ❷
 <text style="header-button">
  <behavior trigger="press" action="back" /> ❸
  Back
 </text>
{% endblock %}

{% block content %} ❹
<view style="details">
 <text style="contact-name">{{ contact.first }} {{ contact.last }}</text>

 <view style="contact-section">
  <text style="contact-section-label">Phone</text>
  <text style="contact-section-info">{{contact.phone}}</text>
 </view>
 <view style="contact-section">
  <text style="contact-section-label">Email</text>
  <text style="contact-section-info">{{contact.email}}</text>
 </view>
</view>
{% endblock %}
```

❶ Расширяет базовый шаблон layout.

❷ Переопределяет блок header шаблона layout, включая в него кнопку Back (Назад).

❸ Поведение для перехода к предыдущему экрану при нажатии.

❹ Переопределяет блок контента для вывода полной информации о выбранном контакте.

Экран с подробной информацией контакта расширяет базовый шаблон layout.xml, как и в index.xml. На этот раз мы переопределяем контент как в блоке заголовка, так и в блоке контента. Переопределение блока заголовка позволяет добавить кнопку возврата с поведением. При нажатии клиент Hyperview раскручивает стек навигации и возвращает пользователю список контактов.

Обратите внимание: инициирование этого поведения — не единственный способ возврата назад. Клиент Hyperview соблюдает соглашения навигации на разных платформах. Пользователи iOS также могут переходить к предыдущему экрану жестом смахивания справа налево, а пользователи Android — нажав физическую кнопку возврата. Чтобы реализовать эти взаимодействия, нам не придется ничего добавлять в HXML.

Внеся всего несколько простых изменений, мы перешли от одноэкранного приложения к многоэкранному. Обратите внимание: для поддержки нового экрана нам почти ничего не пришлось менять в коде мобильного приложения. А вот в традиционной мобильной разработке добавление экранов может быть доволь-

но сложной задачей. Разработчику приходится создавать новый экран, вставлять его в подходящее место навигационной иерархии и писать код для открытия нового экрана из существующих. В Hyperview достаточно добавить поведение с `action="push"`.

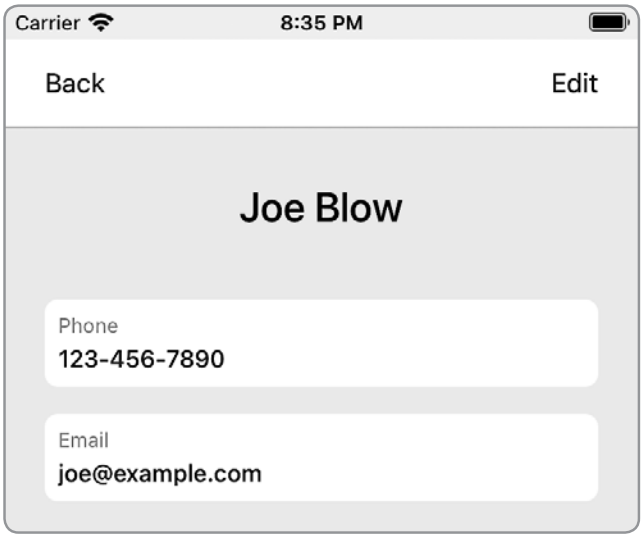


Рис. 16. Экран с подробной информацией о контакте

Редактирование контакта

Сейчас наше приложение позволяет просматривать список контактов и подробную информацию о выбранном контакте. Конечно, было бы полезно иметь возможность обновлять имя, телефон или адрес электронной почты контакта. Следующим улучшением мы добавим интерфейс для редактирования контактов.

Сначала необходимо определить, как именно должен отображаться пользовательский интерфейс редактирования. Можно протолкнуть новый экран редактирования в стек, по аналогии с тем, как это делалось с экраном контактной информации. Однако это не лучший вариант с точки зрения UX. Проталкивание новых экранов в стек имеет смысл при повышении детализации, например при переходе от списка к одному элементу. Однако редактирование не является взаимодействием, повышающим детализацию; это всего лишь переход от просмотра к новому режиму. Соответственно, вместо проталкивания нового экрана в стек навигации мы заменим текущий экран пользовательским интерфейсом редактирования. Для этого нужно добавить кнопку и поведение, использующее действие `reload`. Кнопку можно добавить в заголовок экрана подробной информации контакта.

Листинг 200. Фрагмент hv/show.xml

```
{% block header %}
 <text style="header-button">
  <behavior trigger="press" action="back" />
  Back
 </text>

 <text style="header-button"> ❶
  <behavior trigger="press" action="reload"
href="/contacts/{{contact.id}}/edit" /> ❷
  Edit
 </text>
{% endblock %}
```

❶ Новая кнопка Edit (Редактировать).

❷ Поведение перезагружает текущий экран экраном редактирования при нажатии.

И снова мы повторно используем существующий маршрут Flask (`/contacts/<contact_id>/edit`) для интерфейса редактирования и заполняем идентификатор контакта, используя данные, переданные шаблону Jinja. Также необходимо обновить представление `contacts_edit_get()` чтобы оно возвращало ответ XML на основании шаблона HXML (hv/edit.xml). Мы не приводим пример кода, потому что изменения аналогичны тем, что мы делали для `contacts_view()` в предыдущем разделе. Вместо этого сосредоточимся на шаблоне экрана редактирования.

Листинг 201. hv/edit.xml

```
{% extends 'hv/layout.xml' %}

{% block header %}
 <text style="header-button">
  <behavior trigger="press" action="back" href="#" />
  Back
 </text>
{% endblock %}

{% block content %}
<form> ❶
 <view id="form-fields"> ❷
  {% include 'hv/form_fields.xml' %} ❸
 </view>

 <view style="button"> ❹
  <behavior
   trigger="press"
   action="replace-inner"
   target="form-fields"
   href="/contacts/{{contact.id}}/edit"
   verb="post"
  />
  <text style="button-label">Save</text>
```

```
 </view>
</form>
{% endblock %}
```

❶ Форма с полями ввода и кнопками.
❷ Контейнер с идентификатором, содержащий поля ввода.
❸ Включение шаблона для рендеринга полей ввода.
❹ Кнопка для отправки данных формы и обновления контейнера с полями ввода.

Так как экран редактирования должен отправлять данные на бэкенд, вся секция контента упаковывается в элемент `<form>`. Тем самым гарантируется, что поля данных форм будут включаться в запросы HTTP на бэкенд. В элементе `<form>` пользовательский интерфейс разделен на две части: поля формы и кнопки Save (Сохранить).

Фактические поля формы определяются в отдельном шаблоне (form_fields.xml) и добавляются на экран редактирования с помощью тега Jinja `include`.

Листинг 202. hv/form_fields.xml

```
<view style="edit-group">
 <view style="edit-field">
  <text-field name="first_name" placeholder="First name" value="{{
contact.first }}" /> ❶
  <text style="edit-field-error">{{ contact.errors.first }}</text> ❷
 </view>

 <view style="edit-field"> ❸
  <text-field name="last_name" placeholder="Last name" value="{{
contact.last }}" />
  <text style="edit-field-error">{{ contact.errors.last }}</text>
 </view>
 <!-- Одинаковая разметка для contact.email и contact.phone -->
</view>
```

❶ Текстовое поле с текущим значением имени контакта.
❷ Текстовый элемент, выводящий ошибки модели контакта.
❸ Другое текстовое поле, на этот раз для фамилии контакта.

Мы опустили код для телефона и адреса электронной почты контакта, поскольку он аналогичен коду для имени и фамилии. Каждое поле контакта имеет собственный элемент `<text-field>`, а находящийся под ним элемент `<text>` используется для вывода возможных сообщений об ошибках. `<text-field>` имеет два важных атрибута:

- `name` определяет имя, которое должно использоваться для сериализации значения `text-field` в данные формы для запросов HTTP. Мы используем те же имена, что и в веб-приложении из предыдущих глав (`first_name`,

`last_name`, `phone`, `email`). Так нам не потребуется вносить изменения на бэкенд для парсинга данных формы;

- `value` определяет данные для предварительного заполнения текстовых полей. Так как мы редактируем существующий контакт, разумно заранее заполнить текстовые поля текущим именем, телефоном или адресом электронной почты.

Возникает вопрос: почему мы решили определить поля формы в отдельном шаблоне (form_fields.xml)? Чтобы понять причины такого решения, необходимо сначала обсудить кнопку Save. При нажатии этой кнопки клиент Hyperview выдает запрос HTTP `POST` к `contacts/<contact_id>/edit` с данными формы, сериализованными содержимым элементов ввода `<text-field>`. Ответ HXML заменит содержимое контейнера поля формы (идентификатор `form-fields`). Но каким должен быть этот ответ? Это зависит от результата проверки данных формы.

1. Если данные недопустимы (например, повторяющийся адрес электронной почты), пользовательский интерфейс останется в режиме редактирования, а для недопустимых полей будут выведены сообщения об ошибках. Это позволит пользователю исправить ошибки и повторить попытку сохранения.
2. Если данные допустимы, бэкенд сохранит правки, а пользовательский интерфейс переключится в режим вывода (интерфейс подробной информации о контакте).

Таким образом, бэкенд должен различать допустимые и недопустимые правки. Для поддержки этих двух сценариев внесем некоторые изменения в существующее представление `contacts_edit_post()` в приложении Flask.

Листинг 203. app.py

```
@app.route("/contacts/<contact_id>/edit", methods=["POST"])
def contacts_edit_post(contact_id=0):
  c = Contact.find(contact_id)
  c.update(request.form['first_name'], request.form['last_name'],
request.form['phone'], request.form['email']) ❶
  if c.save(): ❷
    flash("Updated Contact!")
    return render_to_response("hv/form_fields.xml", contact=c,
saved=True) ❸
  else:
    return render_to_response("hv/form_fields.xml", contact=c) ❹
```

❶ Обновляет объект контакта по данным формы запроса.

❷ Пытается сохранить обновления. Возвращает False для недопустимых данных.

❸ При успехе рендерит шаблон полей формы и передает флаг saved шаблону.

❹ При неудаче рендерит шаблон полей формы. В объект контакта добавляются сообщения об ошибках.

Представление уже содержит условную логику в зависимости от того, завершился ли успехом вызов `save()` модели контакта. Если `save()` завершается неудачей, рендерится шаблон form_fields.xml. `contact.errors` будет содержать сообщения об ошибках для недействительных полей, которые будут рендериться в элементах `<text style="edit-field-error">`. Если `save()` завершается успехом, также рендерится шаблон form_fields.xml, но на этот раз шаблон получает флаг `saved` — признак успеха. Обновим шаблон, чтобы этот флаг использовался для реализации нужного поведения: переключения UI обратно в режим вывода.

Листинг 204. hv/form_fields.xml

```
<view style="edit-group">
 {% if saved %} ❶
  <behavior
   trigger="load" ❷
   action="reload" ❸
   href="/contacts/{{contact.id}}" ❹
  />
 {% endif %}

 <view style="edit-field">
  <text-field name="first_name" placeholder="First name" value="{{
contact.first }}" />
  <text style="edit-field-error">{{ contact.errors.first }}</text>
 </view>

 <!-- Та же разметка для других полей -->
</view>
```

❶ Это поведение добавляется только после успешного сохранения контакта.

❷ Поведение срабатывает немедленно.

❸ Поведение перезагружает весь экран.

❹ Экран будет перезагружен экраном подробной информации о контакте.

Шаблон Jinja при помощи условной конструкции проверяет, что поведение будет осуществлять рендеринг только при успешном сохранении, а не при первом открытии экрана (или отправке пользователем недопустимых данных). В случае успеха шаблон добавляет поведение, которое срабатывает немедленно благодаря `trigger="load"`. Действие перезагружает текущий экран экраном подробной информации о контакте (из маршрута `/contacts/<contact_id>`).

Результат? Когда пользователь нажимает кнопку Save, бэкенд сохраняет новые данные контакта, а экран переключается обратно на экран подробной информации (Contact Details). Так как приложение выдает новый запрос HTTP для получения подробной информации о контакте, в нем гарантированно будут отображаться последние сохраненные правки.

ПОЧЕМУ НЕ ПЕРЕНАПРАВЛЕНИЕ?

Возможно, вы помните, что версия этого кода в веб-приложении работала несколько иначе. При успешном сохранении представление возвращало `redirect("/contacts/" + str(contact_id))`. Это перенаправление HTTP дает команду веб-браузеру перейти к странице с подробной информацией о контакте.

В Hyperview такой подход не поддерживается. Почему? В веб-приложении стек навигации имеет простую структуру: это линейная последовательность страниц, в которой активна только одна страница. В мобильных приложениях навигация устроена намного сложнее. Мобильные приложения используют вложенную иерархию стеков навигации, модальных окон и вкладок. Все экраны этой иерархии активны и могут отображаться мгновенно в ответ на действия пользователя. И как в этих условиях клиент Hyperview должен интерпретировать перенаправление HTTP? Должен ли он перезагрузить текущий экран, протолкнуть новый или перейти к экрану в стеке с таким же URL?

Вместо того чтобы принимать решение, которое будет субоптимальным во многих сценариях, Hyperview действует иначе. Перенаправления под управлением сервера невозможны, но внутренняя часть может рендерить поведения навигации в HXML. Именно это мы и делаем, чтобы переключиться с пользовательского интерфейса редактирования на интерфейс подробной информации в приведенном выше коде. Считайте это своего рода перенаправлением на стороне клиента, а еще лучше – навигацией на стороне клиента.

Теперь в приложении для управления контактами есть работоспособный UI редактирования. Чтобы войти в режим редактирования, пользователь нажимает кнопку на экране с подробной информацией о контакте. В режиме редактирования он может обновить данные контакта и сохранить их на бэкенд. Если бэкенд отклоняет изменения как недопустмиые, приложение остается в режиме редактирования и выводит сообщение об ошибках проверки. Если бэкенд принимает и сохраняет изменения, приложение возвращается в режим вывода подробной информации с обновленными данными контактов.

Добавим еще одно улучшение в UI редактирования. Было бы удобно разрешить пользователю выйти из режима редактирования без необходимости сохранять контакт. Обычно для этого предоставляется действие отмены (Cancel). Добавим его в виде новой кнопки под кнопкой Save (Сохранить).

Листинг 205. Фрагмент hv/edit.xml

```
<view style="button">
 <behavior trigger="press" action="replace-inner" target="form-fields"
href="/contacts/{{contact.id}}/edit" verb="post" />
```

```
 <text style="button-label">Save</text>
</view>
<view style="button"> ❶
 <behavior
  trigger="press"
  action="reload" ❷
  href="/contacts/{{contact.id}}" ❸
 />
 <text style="button-label">Cancel</text>
</view>
```

❶ Новая кнопка Cancel (Отменить) на экране редактирования.
❷ При нажатии перезагружает весь экран.
❸ Экран перезагружается экраном подробной информации о контакте.

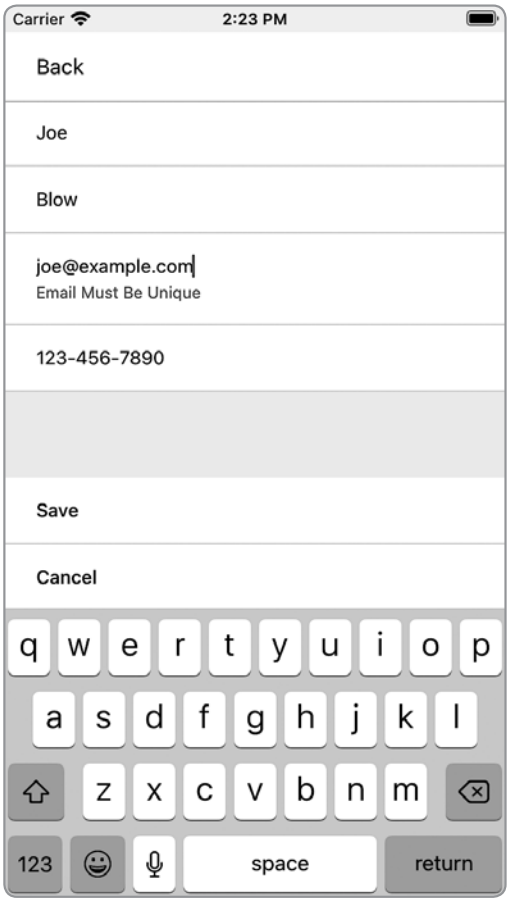


Рис. 17. Экран редактирования контакта

Это тот же прием, который использовался для переключения из UI редактирования на UI подробной информации после успешного редактирования контакта. Однако нажатие Cancel обновляет интерфейс быстрее, чем нажатие Save. При сохранении приложение сначала выдает запрос `POST` для сохранения данных, а затем запрос `GET` для экрана подробной информации. При отмене запрос `POST` пропускается и сразу выдается запрос `GET`.

Обновление списка контактов

Можно считать, что реализация UI редактирования полностью готова. Однако существует одна проблема. Если на этом остановиться, пользователи будут думать, что приложение работает неправильно! Почему? Дело в синхронизации состояния приложения между экранами. Рассмотрим следующую серию взаимодействий.

1. Пользователь запускает приложение в режиме списка контактов.
2. Пользователь нажимает строку «Joe Blow», чтобы загрузить подробную информацию о контакте.
3. Пользователь нажимает кнопку Edit, чтобы переключиться в режим редактирования, и меняет имя контакта на Joseph.
4. Пользователь нажимает кнопку Save, чтобы сохранить данные и вернуться в режим просмотра. Теперь в описании контакта указаны данные Joseph Blow.
5. Пользователь нажимает кнопку Back, чтобы вернуться к списку контактов.

Заметили, что произошло? Список имен остался тем же, как при запуске приложения. Контакт, которому мы только что присвоили имя Joseph, по-прежнему отображается в списке с именем Joe. Эта проблема часто встречается в приложениях гипермедиа. У клиента отсутствует концепция совместного использования данных между разными частями UI. Обновления в одной части приложения не будут автоматически применяться к другим частям приложения.

К счастью, в Hyperview существует решение этой проблемы: *события*. События, встроенные в систему поведений, позволяют упростить коммуникации между разными частями UI.

СОБЫТИЯ И ПОВЕДЕНИЯ

События – клиентский механизм Hyperview. В главе «Скрипты на стороне клиента» рассматривались события при работе с HTML, _hyperscript и DOM. Элементы DOM отправляют события в результате пользовательских взаимодействий. Скрипты могут прослушивать эти события и реагировать на них выполнением произвольного кода JavaScript.

События Hyperview устроены намного проще, они не требуют скриптов и могут объявляться декларативно в HXML. Это делается в системе поведений. Для использования событий необходимо указать новый атрибут поведения, тип действия и тип триггера.

- `event-name`: этот атрибут `<behavior>` определяет имя отправляемого или прослушиваемого события.
- `action="dispatch-event"`: при инициировании это поведение будет отправлять событие с именем, определенным атрибутом `event-name`. Это событие отправляется глобально в пределах всего приложения Hyperview.
- `trigger="on-event"`: это поведение инициируется, если другое поведение в приложении отправляет событие, соответствующее атрибуту `event-name`.

Если элемент `<behavior>` использует `action="dispatch-event"` или `trigger="on-event"`, он также должен определять `event-name`. Учтите, что несколько поведений могут отправлять события с одинаковыми именами. Точно так же несколько поведений могут срабатывать по одному имени события.

Рассмотрим следующее простое поведение:

```
<behavior trigger="press" action="toggle" target="container" />.
```

Нажатие элемента, содержащего это поведение, переключает видимость элемента с идентификатором `container`. Но что, если элемент, который нужно переключить, находится на другом экране? Действие `toggle` и поиск идентификатора цели работают только на текущем экране, так что это решение не годится. Проблема решается созданием двух поведений, по одному для каждого экрана, взаимодействующих через механизм событий.

- Экран A: `<behavior trigger="press" action="dispatch-event" event-name="button-pressed" />`
- Экран B: `<behavior trigger="on-event" event-name="button-pressed" action="toggle" target="container" />`

Нажатие элемента, содержащего первое поведение (на экране A), отправляет событие с именем `button-pressed`. Второе поведение (на экране B) инициируется по событию с этим именем и переключает состояние видимости элемента с идентификатором `container`.

События находят множество практических применений, но самое распространенное из них – оповещение разных экранов об изменениях в состоянии бэкенда, требующих перезагрузки пользовательского интерфейса.

Вы уже достаточно знаете о системе событий Hyperview, чтобы устранить ошибку в приложении. Когда пользователь сохраняет изменение в контакте, необходимо отправить событие с экрана подробной информации. А экран контактов должен прослушать это событие и перезагрузиться, чтобы отреагировать на правку. Так как шаблон form_fields.xml уже получает флаг `saved` при успешном сохранении контакта на бэкенд, событие будет удобно отправить из этой точки.

Листинг 206. Фрагмент hv/form_fields.xml

```
{% if saved %}
 <behavior
  trigger="load" ❶
  action="dispatch-event" ❷
  event-name="contact-updated" ❸
 />
 <behavior ❹
  trigger="load"
  action="reload"
  href="/contacts/{{contact.id}}"
 />
{% endif %}
```

❶ Инициирует поведение немедленно.
❷ Поведение отправляет событие.
❸ Событию присваивается имя contact-updated.
❹ Существующее поведение для вывода UI подробной информации о контакте.

Теперь необходимо сделать так, чтобы список контактов прослушивал событие `contact-updated` и перезагружался:

Листинг 207. Фрагмент hv/index.xml

```
<form>
 <behavior
  trigger="on-event" ❶
  event-name="contact-updated" ❷
  action="replace-inner" ❸
  target="contacts-list"
  href="/contacts?rows_only=true"
  verb="get"
 />
 <!-- Разметка text-field опущена -->
 <list id="contacts-list">
  {% include 'hv/rows.xml' %}
 </list>
</form>
```

❶ Инициирует поведение при отправке события.

❷ Инициирует поведение для отправленных событий с именем contact-updated.
❸ При срабатывании заменяет содержимое элемента <list> строками, полученными от бэкенда.

Каждый раз, когда пользователь редактирует контакт, экран со списком контактов обновляется в соответствии с внесенными изменениями. Добавление этих двух элементов `<behavior>` исправляет ошибку: на экране со списком контактов в списке отображается верное имя «Joseph Blow». Обратите внимание: мы намеренно добавили новое поведение в элемент `<form>`. Это гарантирует, что инициированный запрос сохранит запрос поиска.

Чтобы показать, что мы имеем в виду, вернемся к последовательности действий, демонстрирующих ошибочное поведение. Предположим, что перед тем как нажать контакт Joe Blow, пользователь провел поиск в контактах, для чего ввел Joe в поле поиска. Когда пользователь обновляет контакт именем Joseph Blow, шаблон отправляет событие `contact-updated`, в результате чего инициируется поведение `replace-inner` для экрана со списком контактов. Из-за присутствия родительского элемента `<form>` поиск Joe будет сериализован в запрос `GET /contacts?rows_only=true&q=Joe`. Так как имя Joseph не соответствует критерию Joe, отредактированный контакт не появится в списке (пока пользователь не очистит запрос). Состояние приложения остается согласованным между бэкендом и всеми активными экранами.

События вводят в поведение дополнительный уровень абстракции. До сих пор мы видели, что редактирование контакта вызывает обновление списка контактов. Однако список контактов должен обновляться и после других действий, например удаления или добавления контакта. При условии, что ответы HXML для удаления или создания включают поведение отправки события `contact-updated`, мы получим желаемое поведение экрана со списком контактов.

Экран не интересует, что привело к отправке события `contact-updated`. Он просто знает, что нужно делать, когда это происходит.

Удаление контакта

Раз уж мы упомянули об удалении контакта, займемся реализацией этой функции. Пользователь сможет удалить контакт из пользовательского интерфейса редактирования. Добавим новую кнопку в файл edit.xml.

Листинг 208. Фрагмент hv/edit.xml

```
<view style="button">
 <behavior trigger="press" action="replace-inner" target="form-fields"
```

```
href="/contacts/{{contact.id}}/edit" verb="post" />
 <text style="button-label">Save</text>
</view>
<view style="button">
 <behavior trigger="press" action="reload" href="/contacts/{{contact.id}}"
/>
 <text style="button-label">Cancel</text>
</view>
<view style="button"> ❶
 <behavior
  trigger="press"
  action="append" ❷
  target="form-fields"
  href="/contacts/{{contact.id}}/delete" ❸
  verb="post"
 />
 <text style="button-label button-label-delete">Delete Contact</text>
</view>
```

❶ Новая кнопка Delete Contact (Удалить контакт) на экране редактирования.
❷ По нажатию присоединяет HXML к контейнеру на экране.
❸ HXML загружается по запросу POST /contacts/<contact_id>/delete.

Разметка HXML для кнопки Delete (Удалить) напоминает разметку для кнопки Save (Сохранить), но между ними есть ряд неочевидных различий. Напомним, что нажатие кнопки Save приводит к одному из двух ожидаемых результатов: неудаче с выводом ошибок проверки данных формы или успеху с переключением на экран подробной информации. Для поддержки первого результата (неудачи с выводом ошибок проверки данных) поведение сохранения заменяет содержимое контейнера `<view id="form-fields">` заново отрендеренной версией form_fields.xml. В этом случае использование действия `replace-inner` оправданно.

При удалении фаза проверки данных отсутствует, поэтому возможен только один ожидаемый результат: успешное удаление контакта. Если удаление прошло успешно, контакт перестает существовать. Нет смысла показывать интерфейс редактирования или подробную информацию о несуществующем контакте. Вместо этого приложение возвращается к предыдущему экрану (списку контактов). Ответ включает только поведение, которое срабатывает немедленно, изменения UI не нужны. Таким образом, использование действия `append` сохраняет текущий UI, пока Hyperview выполняет действия.

Листинг 209. Фрагмент hv/deleted.xml

```
<view>
 <behavior trigger="load" action="dispatch-event" event-name="contact-
updated" /> ❶
```

```
 <behavior trigger="load" action="back" /> ❷
</view>
```

❶ При загрузке отправляется событие contact-updated для обновления экрана со списком контактов.

❷ Возвращается к экрану со списком контактов.

Заметим, что, помимо поведения возврата, этот шаблон включает поведение отправки события `contact-updated`. В предыдущем разделе мы добавили в index.xml поведение обновления списка при отправке этого события. Отправляя событие после удаления, мы гарантируем, что контакт будет удален из списка.

И снова мы пропустим изменения бэкенда Flask. Отметим только, что нужно обновить представление `contacts_delete()`, чтобы оно отвечало шаблоном hv/deleted.xml. А еще необходимо обновить маршрут, чтобы он поддерживал `POST` в дополнение к `DELETE`, так как клиент Hyperview понимает только `GET` и `POST`.

В приложении появилась полностью рабочая функция удаления! Впрочем, она не самая удобная: одного случайного касания достаточно, чтобы навсегда удалить контакт. Для деструктивных действий (таких, как удаление контакта) всегда лучше запрашивать у пользователя подтверждение.

Чтобы добавить подтверждение к поведению удаления, можно воспользоваться системным действием `alert`, описанным в предыдущей главе. Напомним, что действие `alert` выведет системное диалоговое окно с кнопками, инициирующими другие варианты поведения. Все, что для этого нужно, — упаковать элемент `<behavior>` для удаления в элемент `<behavior>`, использующий `action="alert"`.

Листинг 210. Кнопка удаления в hv/edit.xml

```
<view style="button">
 <behavior ❶
  xmlns:alert="https://hyperview.org/hyperview-alert"
  trigger="press"
  action="alert"
  alert:title="Confirm delete"
  alert:message="Are you sure you want to delete {{ contact.first }}?"
 >
  <alert:option alert:label="Confirm"> ❷
   <behavior ❸
    trigger="press"
    action="append"
    target="form-fields"
    href="/contacts/{{contact.id}}/delete"
    verb="post"
   />
```

```
  </alert:option>
  <alert:option alert:label="Cancel" /> ❹
 </behavior>
 <text style="button-label button-label-delete">Delete Contact</text>
</view>
```

❶ Нажатие Delete (Удалить) инициирует действие для показа системного диалогового окна с заданным текстом заголовка («Подтвердить удаление») и сообщением («Вы уверены, что хотите удалить...?»).

❷ Первый активный вариант в системном диалоговом окне.

❸ При нажатии первого варианта инициируется удаление контакта.

❹ Второй активный вариант не имеет поведения, поэтому он только закрывает диалоговое окно.

В отличие от предыдущего сценария, нажатие кнопки удаления не производит немедленного эффекта. Вместо этого открывается диалоговое окно с предложением подтвердить или отменить удаление. Базовое поведение удаления не изменилось, оно просто было соединено с другим поведением.

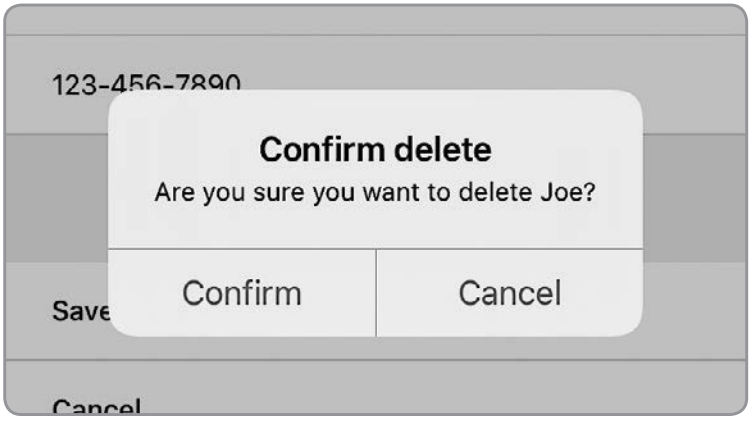


Рис. 18. Окно подтверждения удаления контакта

Добавление нового контакта

Добавление нового контакта — последняя функциональность, которую должно поддерживать наше мобильное приложение. К счастью, она еще и реализуется проще остальных. При этом можно повторно использовать концепции (и даже некоторые шаблоны) из уже реализованной функциональности. В частности,

добавление нового контакта очень похоже на редактирование существующего. Обе функции должны:

- добавить форму для сбора информации о контакте;
- иметь возможность сохранить введенную информацию;
- выводить ошибки проверки данных формы;
- сохранить контакт при отсутствии ошибок проверки данных.

Так как функциональность очень похожа, мы кратко перечислим изменения, не приводя код.

1. Обновление index.xml.
   - Переопределить блок заголовка, добавив новую кнопку Add (Добавить).
   - Включить поведение в кнопку. При нажатии протолкнуть новый экран в стек в модальном режиме, используя `action="new"`, и запросить содержимое экрана из /contacts/new.
2. Создание шаблона hv/new.xml.
   - Переопределить блок заголовка, добавив кнопку, которая будет закрывать модальное окно, используя `action="close"`.
   - Включить шаблон hv/form_fields.xml, который рендерит пустые поля формы.
   - Добавить кнопку Add Contact (Добавить контакт) под полями формы.
   - Включить поведение в кнопку. При нажатии кнопка должна выдавать запрос `POST` к `/contacts/new`; использовать `action="replace-inner"` для обновления полей формы.
3. Обновление представления Flask.
   - Изменить `contacts_new_get()`, чтобы использовать `render_to_response()` с шаблоном hv/new.xml.
   - Изменить `contacts_new()`, чтобы использовать `render_to_response()` с шаблоном hv/form_fields.xml. Передать `saved=True` при рендеринге шаблона после успешного сохранения нового контакта.

Повторно используя form_fields.xml для редактирования и добавления контакта, мы повторно используем часть кода и обеспечиваем согласование пользовательских интерфейсов двух функций. Кроме того, экран Add Contact сможет воспользоваться логикой `saved`, которая является частью form_fields.xml. После успешного добавления нового контакта экран отправляет событие `contact-updated`, по которому обновляется список контактов и выводится добавленный контакт. Экран перезагружается, чтобы показать подробную информацию о контакте.

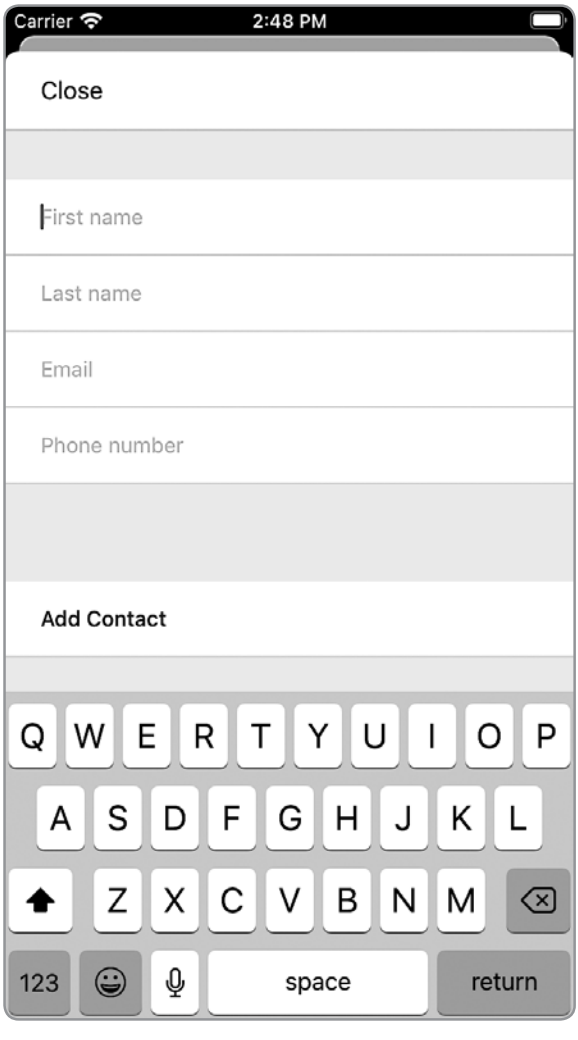


Рис. 19. Модальное окно добавления контакта

Развертывание приложения

После завершения работы над интерфейсом создания контактов мы получаем полностью готовое мобильное приложение. В нем поддерживается поиск по списку контактов, просмотр подробной информации о контакте, редактирование и удаление контактов, а также добавление нового контакта. Но до сих пор мы разрабатывали приложение в симуляторе на десктопном компьютере. Как уви-

деть его работающим на мобильном устройстве? И как распространять его среди пользователей?

Чтобы увидеть, как приложение работает на физическом устройстве, воспользуемся средствами предварительного просмотра платформы Expo.

1. Загрузите приложение Expo Go на устройство Android или iOS.
2. Перезапустите приложение Flask с привязкой к интерфейсу, доступному по локальной сети. Команда выглядит примерно так:

   `flask run --host 192.168.7.229`, где `host` — IP-адрес вашего компьютера в сети.

3. Обновите код клиента Hyperview, чтобы значение `ENTRY_POINT_URL` (в demo/src/constants.js) указывало на IP-адрес и порт, с которыми связан сервер Flask.
4. После выполнения команды `yarn start` в демоприложении Hyperview на консоли появится QR-код с инструкциями о том, как сканировать его на Android и iOS.

Когда вы отсканируете QR-код, приложение запустится на устройстве. Вы увидите, что в процессе взаимодействия с приложением серверу Flask отправляются запросы HTTP. Физическое устройство даже может использоваться в ходе разработки. После каждого изменения в HXML необходимо перезагружать экран, чтобы увидеть обновления интерфейса.

Итак, приложение работает на физическом устройстве, но все еще не готово для реальной эксплуатации. Чтобы приложение попало к пользователям, осталось выполнить несколько действий.

1. Развернуть бэкенд на продакшен. Вместо сервера разработки Flask необходимо использовать рабочий веб-сервер, например Gunicorn. Также приложение должно работать на машине, доступной по Сети (скорее всего, с помощью облачного провайдера, такого как AWS или Heroku).
2. Создать автономные двоичные приложения. Следуя инструкциям из проекта Expo, можно создать файл .ipa или .apk для платформ iOS или Android. Не забудьте обновить значение `ENTRY_POINT_URL` в клиенте Hyperview, чтобы оно ссылалось на бэкенд рабочей версии.
3. Отправить двоичные файлы в iOS App Store или в Google Play и дождаться, когда приложение пройдет проверку.

Приложение одобрено — поздравляем! Теперь его могут загружать пользователи Android и iOS. А самое замечательное, так как в приложении используется архитектура гипермедиа, можно расширять его функциональность простым обновлением бэкенда. Пользовательский интерфейс и взаимодействия полно-

стью определяются в разметке HXML, генерируемой шаблонами на стороне сервера. Хотите добавить на экран новый раздел? Просто обновите существующий шаблон HXML. Хотите добавить новый экран в приложение? Создайте новый маршрут, представление и шаблон HXML. Затем добавьте к существующему экрану поведение, которое будет открывать новый экран. Чтобы доставить эти изменения пользователям, достаточно заново развернуть бэкенд. Приложение знает, как интерпретировать HXML, и этого достаточно, чтобы работать с новой функциональностью.

Один бэкенд, несколько форматов гипермедиа

Чтобы создать мобильное приложение на основе архитектуры гипермедиа, мы начали с веб-приложения и внесли ряд изменений, самым важным из которых была замена шаблонов HTML шаблонами HXML. Однако в процессе портирования бэкенда в мобильное приложение функциональность веб-приложения была нарушена. Попытавшись открыть адрес http://0.0.0.0:5000 в веб-браузере, вы бы увидели беспорядочный набор текста и разметки XML. Дело в том, что веб-браузеры не умеют рендерить простую разметку XML и тем более не знают, как интерпретировать теги и атрибуты HXML для рендеринга приложения. И это печально, поскольку код Flask для веб-приложения и мобильного приложения практически идентичен. В нем используется общая логика базы данных и модели, а большинство представлений также остается без изменений.

Наверняка у вас появился логичный вопрос: можно ли использовать один бэкенд как для веб-приложения, так и для мобильного приложения? Да, можно! Собственно, это одно из преимуществ использования архитектуры гипермедиа на разных платформах. Вам не придется портировать логику на стороне клиента с одной платформы на другую, достаточно отвечать на запросы в соответствующем формате. Для этого воспользуемся механизмом *согласования контента*, встроенным в HTTP.

Что такое согласование контента?

Представьте, что два пользователя, носители немецкого и японского языков, открывают сайт https://google.com в своем браузере. Они видят домашнюю страницу Google, локализованную для немецкого и японского языков соответственно. Как Google определяет, что нужно вернуть разные версии домашней страницы в зависимости от языковых предпочтений пользователя? Ответ кроется в архитектуре REST и в том, как в ней разделяются концепции ресурсов и представлений.

В архитектуре REST домашняя страница Google рассматривается как единый «ресурс», представляемый уникальным URL. Однако этот один ресурс может иметь много «представлений», то есть вариантов представления ресурса для клиента. Японская и немецкая версии домашней страницы Google являются двумя представлениями одного ресурса. Чтобы определить лучшее представление для возвращаемого ресурса, клиенты и серверы HTTP участвуют в процессе, называемом «согласованием контента» (content negotiation). Он работает следующим образом:

- клиенты задают предпочтительное представление в заголовках запроса `Accept-*`;
- сервер подбирает оптимальный вариант предпочтительного представления и возвращает выбранное представление с использованием заголовка `Content-*`.

В примере с домашней страницей Google немецкоговорящий пользователь использует браузер, в котором настроен предпочтительный контент, локализованный для немецкого языка. Каждый запрос HTTP, выдаваемый веб-браузером, включает заголовок `Accept-Language: de-DE`. Сервер видит заголовок запроса и возвращает ответ, локализованный для немецкого языка (если это возможно). Ответ HTTP включает заголовок `Content-Language: de-DE`, сообщающий клиенту язык контента ответа.

Язык — всего лишь один из факторов представления ресурсов. Что еще важнее, ресурсы могут представляться с использованием разных типов контента, таких как HTML или HXML. Согласование по типу контента осуществляется на основании заголовков запросов `Accept` и заголовков ответов `Content-Type`. Веб-браузеры назначают `text/html` как предпочтительный тип контента в заголовке `Accept`. Клиент Hyperview задает `application/vnd.hyperview+xml` как предпочтительный тип контента. Так бэкенд получает возможность различать запросы, поступающие от веб-браузера или клиента Hyperview, и предоставлять каждому соответствующий контент.

Существуют два основных способа согласования контента: детализированный и глобальный.

Способ 1. Переключение шаблонов

Когда мы портировали приложение Contacts с веб-платформы на мобильную, мы сохранили все представления Flask, но внесли в них ряд незначительных изменений, а именно добавили новую функцию `render_to_response()` и вызвали ее в команде `return` каждого представления. Напомним вам эту функцию.

Листинг 211. app.py

```
def render_to_response(template_name, *args, **kwargs):
  content = render_template(template_name, *args, **kwargs)
  response = make_response(content)
  response.headers['Content-Type'] = 'application/vnd.hyperview+xml'
  return response
```

`render_to_response()` рендерит шаблон с заданным контекстом и преобразует его в объект ответа Flask с соответствующим заголовком Hyperview `Content-Type`. Очевидно, реализация адаптирована для мобильного приложения Hyperview. Но мы можем изменить функцию для согласования контента на основании заголовка `Accept` запроса.

Листинг 212. app.py

```
HTML_MIME = 'text/html'
HXML_MIME = 'application/vnd.hyperview+xml'

def render_to_response(html_template_name, hxml_template_name, *args,
**kwargs): ❶
  response_type = request.accept_mimetypes.best_match([HTML_MIME,
HXML_MIME], default=HTML_MIME) ❷
  template_name = hxml_template_name if response_type == HXML_MIME else
html_template_name ❸
  content = render_template(template_name, *args, **kwargs)
  response = make_response(content)
  response.headers['Content-Type'] = response_type ❹
  return response
```

❶ Сигнатура функции получает два шаблона, для HTML и HXML.

❷ Определяет, что нужно клиенту — HTML или HXML.

❸ Выбирает шаблон, который лучше подходит клиенту.

❹ Задает заголовок Content-Type в зависимости от варианта, который лучше подходит клиенту.

Для упрощения согласования объект запроса Flask предоставляет свойство `accept_mimetypes`. Мы передаем `request.accept_mimetypes.best_match()` два типа контента MIME и получаем обратно тип MIME, подходящий для клиента. В зависимости от типа MIME рендерится либо шаблон HTML, либо шаблон HXML. Также необходимо присвоить заголовку `Content-Type` соответствующий тип MIME. Единственное отличие в представлениях Flask заключается в том, что мы должны предоставить как шаблон HTML, так и шаблон HXML.

Листинг 213. app.py

```
@app.route("/contacts/<contact_id>")
def contacts_view(contact_id=0):
  contact = Contact.find(contact_id)
  return render_to_response("show.html", "hv/show.xml", contact=contact) ❶
```

❶ Переключается между шаблонами HTML и HXML в зависимости от клиента.

После обновления всех представлений Flask для поддержки обоих шаблонов бэкенд будет поддерживать как веб-браузеры, так и мобильное приложение! Этот прием отлично работает в приложении Contacts, потому что экраны мобильного приложения напрямую соотносятся со страницами веб-приложения. В обоих приложениях есть специальная страница (или экран) для списка контактов, вывода и редактирования подробной информации и создания нового контакта. Это означает, что представления Flask можно использовать как есть, без серьезных изменений.

Но что, если понадобится переопределить пользовательский интерфейс для мобильного приложения? Допустим, в мобильном приложении вы хотите использовать один экран со строками, которые будут разворачиваться, чтобы давать возможность просматривать и редактировать информацию? Когда на разных платформах разный UI, переключение шаблонов становится неудобным или невозможным. Чтобы сохранить один бэкенд для обоих форматов гипермедиа, придется поискать другой подход.

Способ 2. Выбор перенаправления

Вероятно, вы помните, что в веб-приложении Contacts существует представление `index` с корневым путем `/`.

Листинг 214. app.py

```
@app.route("/")
def index():
  return redirect("/contacts") ❶
```

❶ Перенаправляет запросы от «/» к «/contacts».

При поступлении запроса к корневому пути веб-приложения Flask перенаправляет его по пути `/contacts`. Перенаправление работает и в мобильном приложении Hyperview. Параметр `ENTRY_POINT_URL` клиента Hyperview указывает на http://0.0.0.0:5000/, а сервер перенаправляет запросы к http://0.0.0.0:5000/contacts.

Но выполнять перенаправления по одному пути в веб-приложении и в мобильном приложении вовсе не обязательно. Что, если воспользоваться заголовком `Accept` для выбора пути перенаправления?

Листинг 215. app.py

```
HTML_MIME = 'text/html'
HXML_MIME = 'application/vnd.hyperview+xml'

@app.route("/")
def index():
  response_type = request.accept_mimetypes.best_match([HTML_MIME,
HXML_MIME], default=HTML_MIME) ❶
  if response_type == HXML_MIME:
   return redirect("/mobile/contacts") ❷
  else:
   return redirect("/web/contacts") ❸
```

❶ Определяет, что запрашивает клиент — HTML или HXML.
❷ Если клиент запрашивает HXML, он перенаправляется к /mobile/contacts.
❸ Если клиент запрашивает HTML, он перенаправляется к /web/contacts.

Точка входа превращается в развилку: если клиенту нужна разметка HTML, он направляется на один путь, а если разметка HXML — на другой путь. Эти перенаправления будут обрабатываться разными представлениями Flask:

Листинг 216. app.py

```
@app.route("/mobile/contacts")
def mobile_contacts():
 # Рендерит ответ HXML

@app.route("/web/contacts")
def web_contacts():
 # Рендерит ответ HTML
```

Представление `mobile_contacts()` рендерит шаблон HXML со списком контактов. Когда пользователь касается одного из контактов, открывается экран, запрашиваемый по пути `/mobile/contacts/1`, который обрабатывается представлением `mobile_contacts_view`. После исходного ветвления все последующие запросы от мобильного приложения передаются путям с префиксом `/mobile/` и обрабатываются представлениями Flask для мобильных устройств.

Точно так же все последующие запросы от веб-приложения передаются путям с префиксом `/web/` и обрабатываются представлениями Flask для веб-приложений. (В реальном приложении представления для веб- и мобильных приложений следовало бы разделить по разным частям кода: web_app.py

и mobile_app.py. Возможно, также стоило бы отказаться от использования префикса /web/ в веб-путях, чтобы URL в адресной строке браузера выглядели более элегантно.)

Возможно, вы думаете, что выбор перенаправления приводит к большому объему дублированного кода. В конце концов, придется писать вдвое больше представлений: для веб-приложения и мобильного приложения. И это правда, поэтому выбор перенаправления делается только тогда, когда две платформы требуют изолирования групп логики представлений. Если приложения имеют похожую структуру на обеих платформах, переключение шаблонов сэкономит немало времени и обеспечит согласованность. Даже при использовании решения с выбором перенаправления большая часть логики моделей может совместно использоваться обоими наборами представлений.

Может случиться так, что вы начнете использовать переключение шаблонов, а затем поймете, что вам необходимо реализовать выбор перенаправления для специфической функциональности конкретных платформ. Собственно, мы уже делали так в приложении Contacts. При портировании веб-приложения в мобильное мы не стали переносить некоторые возможности, например функциональность архивации. UI динамической архивации — мощный механизм, который не имеет смысла на мобильном устройстве. Так как наши шаблоны HXML не предоставляют никаких точек входа к функциональности архивации, можно считать, что эта функциональность актуальна только для веб-среды, и не беспокоиться о ее поддержке в Hyperview.

Contact.app в Hyperview

В этой главе мы узнали довольно много нового. Переведите дух и вспомните, какой путь мы прошли: все началось с портирования базовой функциональности веб-приложения Contact.app в мобильную версию. Для этого нам удалось повторно использовать большую часть кода внутренней части Flask, не отказываясь от шаблонов Jinja. Также мы снова продемонстрировали полезность событий для связывания разных частей приложения.

Впрочем, это еще не все. В следующей главе мы реализуем нестандартное поведение и элементы UI, чтобы завершить работу над мобильным приложением Contact.app.

Заметки о гипермедиа: конечные точки API

В отличие от JSON API, API гипермедиа, создаваемый для гипермедиа-управляемых приложений, должен предоставлять конечные точки, специализированные для потребностей UI конкретного приложения.

Так как API гипермедиа не рассчитаны на потребление клиентами общего назначения, можно не заботиться о поддержании их в обобщенном виде и генерировать контент, предназначенный именно для создаваемого приложения. Конечные точки должны быть оптимизированы для поддержки потребностей в UI/UX конкретного приложения, а не для обобщенного доступа к данным модели предметной области.

Отсюда следует один совет: используя API на основе гипермедиа, можно провести агрессивный рефакторинг API способом, который крайне не рекомендуется применять в приложениях SPA на основе JSON API или в мобильных клиентах. Так как приложения на основе гипермедиа используют принцип HATEOAS («гипермедиа как ядро состояния приложения»), можно (и более того, нужно!) изменять их конфигурацию по мере развития приложения и изменения сценариев использования.

Огромное преимущество подхода гипермедиа заключается в том, что можно полностью переработать API, чтобы адаптироваться к новым потребностям со временем без необходимости версионировать API и даже документировать его. Пользуйтесь этой возможностью по максимуму!

РАСШИРЕНИЕ КЛИЕНТА HYPERVIEW

В предыдущей главе мы создали полнофункциональную мобильную версию приложения Contacts. Не считая настройки конечной точки URL, мы не изменяли никакой код, выполняемый на мобильном устройстве. Весь пользовательский интерфейс и логика мобильного приложения были полностью определены в коде бэкенда с использованием Flask и шаблонов HXML. Это стало возможным, поскольку стандартный клиент Hyperview поддерживает весь основной функционал мобильных приложений.

Однако стандартный клиент Hyperview не может сделать за вас все и сразу. Разработчикам нужно, чтобы их приложения имели нестандартные средства UI или глубоко интегрировались с нативной функциональностью. Для удовлетворения этих потребностей при проектировании клиента Hyperview в него были заложены возможности расширения нестандартными действиями и элементами UI. В этом разделе мы дополним наше мобильное приложение примерами из обеих категорий.

Но прежде чем браться за дело, кратко рассмотрим технологический стек, который будет использоваться. Клиент Hyperview написан на основе React Native — популярного мобильного кросс-платформенного фреймворка. Он использует такой же API на основе компонентов, что и React. Это означает, что разработчики, знакомые с JavaScript и React, быстро освоят React Native. Для React Native существует развитая экосистема библиотек с открытым исходным кодом. Мы будем пользоваться этими библиотеками для создания собственных нестандартных расширений клиента Hyperview.

Добавление функций телефонных звонков и отправки электронной почты

Начнем с самой очевидной функциональности, отсутствующей в приложении Contacts: телефонных звонков. Мобильные устройства могут совершать телефонные звонки. У контактов в приложении есть телефонные номера. Почему

бы приложению не поддерживать звонки по этим номерам? И раз на то пошло, приложение также могло бы отправлять контактам сообщения электронной почты.

В веб-среде для телефонных звонков поддерживается схема URI `tel:`, а для отправки электронной почты — схема URI `mailto:`.

Листинг 217. Схемы tel и mailto в HTML

```
<a href="tel:555-555-5555">Call</a> ❶
<a href="mailto:joe@example.com">Email</a> ❷
```

❶ При нажатии предлагает пользователю позвонить по заданному номеру.
❷ При нажатии открывает клиент электронной почты с заданным адресом в поле to:.

Клиент Hyperview не поддерживает схемы URI `tel:` и `mailto:`. Тем не менее можно добавить эти возможности с помощью нестандартных действий поведения. Напомним, что поведениями называются взаимодействия, определенные в HXML. С поведениями связываются триггеры (`press`, `refresh`) и действия (`update`, `share`). Значения действий не ограничиваются набором, включенным в библиотеку Hyperview. Определим два новых действия, `open-phone` и `open-email`.

Листинг 218. Действия для телефонных звонков и отправки электронной почты

```
<view xmlns:comms="https://hypermedia.systems/hyperview/communications"> ❶
 <text>
  <behavior action="open-phone" comms:phone-number="555-555-5555" /> ❷
  Call
 </text>
 <text>
  <behavior action="open-email" comms:email-address="joe@example.com" /> ❸
  Email
 </text>
</view>
```

❶ Определяет псевдоним для пространства имен XML, используемого новыми атрибутами.
❷ При нажатии предлагает пользователю позвонить по заданному номеру.
❸ При нажатии открывает клиент электронной почты с заданным адресом в поле to:.

Обратите внимание: телефон и адрес электронной почты определяются разными атрибутами. В HTML схема и данные упаковываются в атрибут `href`. Элементы `<behavior>` в HXML предоставляют больше вариантов для представления данных. Мы решили использовать атрибуты, но для представления телефона и адреса также можно было воспользоваться дочерними элементами. Также

пространство имен используется для предотвращения потенциальных конфликтов с другими клиентскими расширениями.

Пока все неплохо, но каким образом клиент Hyperview узнает, как интерпретировать `open-phone` и `open-email` и как сослаться на атрибуты `phone-number` и `email-address`? Для этого нам наконец придется написать код JavaScript.

Начнем с того, что в приложение будет добавлена сторонняя библиотека (`react-native-communications`). Библиотека предоставляет простой API, взаимодействующий с функциональностью звонков и отправки электронной почты уровня ОС:

```
> cd hyperview/demo
> yarn add react-native-communications ❶
> yarn start ❷
```

❶ Добавляет зависимость от react-native-communications.
❷ Перезапускает мобильное приложение.

Затем создается новый файл phone.js с кодом, связанным с действием `open-phone`.

Листинг 219. demo/src/phone.js

```
import { phonecall } from 'react-native-communications'; ❶

const namespace = "https://hypermedia.systems/hyperview/communications";

export default {
 action: "open-phone", ❷
 callback: (behaviorElement) => { ❸
  const number = behaviorElement.getAttributeNS(namespace, "phone-number");
❹
  if (number != null) {
   phonecall(number, false); ❺
  }
 },
};
```

❶ Импортирует необходимую функцию из сторонней библиотеки.
❷ Имя действия.
❸ Обратный вызов, срабатывающий при инициировании действия.
❹ Получает телефон из элемента <behavior>.
❺ Передает телефон функции из сторонней библиотеки.

Нестандартные действия определяются в виде объекта JavaScript с двумя ключами: `action` и `callback`. Так клиент Hyperview связывает нестандартное действие

в HXML с нашим специальным кодом. Значение `callback` представляет собой функцию, получающую один параметр `behaviorElement`. Этот параметр является представлением DOM в формате XML элемента `<behavior>`, инициировавшего действие. Это означает, что для него можно вызывать методы (например, `getAttribute`) или обращаться к его атрибутам (например, `childNodes`). В данном случае метод `getAttributeNS` используется для чтения номера телефона из атрибута `phone-number` в элементе `<behavior>`. Если номер телефона определяется в элементе, можно вызвать функцию `phonecall()`, предоставляемую библиотекой `react-native-communications`.

Прежде чем мы сможем использовать свое нестандартное действие, остается сделать еще одно: зарегистрировать действие в клиенте Hyperview. Клиент Hyperview выступает в виде компонента React Native с именем `Hyperview`. Этот компонент получает свойство с именем `behaviors`, которое представляет собой массив объектов нестандартных действий, таких как наше действие `open-phone`. Передадим нашу реализацию `open-phone` компоненту Hyperview в приложении.

Листинг 220. demo/src/HyperviewScreen.js

```
import React, { PureComponent } from 'react';
import Hyperview from 'hyperview';
import OpenPhone from './phone'; ❶

export default class HyperviewScreen extends PureComponent {
 // ... Фрагмент опущен для краткости

 behaviors = [OpenPhone]; ❷

 render() {
  return (
   <Hyperview
    behaviors={this.behaviors} ❸
    entrypointUrl={this.entrypointUrl}
    // Другие свойства...
   />
  );
 }
}
```

❶ Импортирует действие open-phone.

❷ Создает массив нестандартных действий.

❸ Передает нестандартные действия компоненту Hyperview в виде свойства с именем behaviors.

Во внутренней реализации компонент Hyperview отвечает за преобразование HXML в элементы мобильного UI. Он также обеспечивает инициирование

действий поведения на основании пользовательских взаимодействий. Передавая Hyperview действие `open-phone`, можно использовать его как значение атрибута `action` элементов `<behavior>`. Чтобы сделать это, обновим шаблон show.xml в приложении Flask.

Листинг 221. Фрагмент hv/show.xml

```
{% block content %}
<view style="details">
 <text style="contact-name">{{ contact.first }} {{ contact.last }}</text>

 <view style="contact-section">
  <behavior ❶
   xmlns:comms="https://hypermedia.systems/hyperview/communications"
   trigger="press"
   action="open-phone" ❷
   comms:phone-number="{{contact.phone}}" ❸
  />
  <text style="contact-section-label">Phone</text>
  <text style="contact-section-info">{{contact.phone}}</text>
 </view>

 <view style="contact-section">
  <behavior ❹
   xmlns:comms="https://hypermedia.systems/hyperview/communications"
   trigger="press"
   action="open-email"
   comms:email-address="{{contact.email}}"
  />
  <text style="contact-section-label">Email</text>
  <text style="contact-section-info">{{contact.email}}</text>
 </view>
</view>
{% endblock %}
```

❶ Добавляет в раздел с телефоном поведение, инициируемое по триггеру press.

❷ Инициирует новое действие open-phone.

❸ Задает атрибут, который ожидает получить действие open-phone.

❹ Тот же принцип с другим действием (open-email).

Мы опустим реализацию второго нестандартного действия, `open-email`. Как нетрудно предположить, это действие открывает системное приложение для подготовки сообщений, чтобы пользователь мог отправить контакту электронное письмо. Реализация `open-email` почти идентична `open-phone`. Библиотека `react-native-communications` предоставляет функцию с именем `email()`, так что мы можем просто упаковать ее и передать ей аргументы тем же способом.

Теперь у нас имеется завершенный пример расширения клиента нестандартными действиями поведения. Мы выбрали для действий новые имена (`open-phone` и `open-email`) и связали их с функциями. Эти функции получают элементы `<behavior>` и могут выполнять произвольный код React Native. Мы упаковали существующую стороннюю библиотеку и прочитали атрибуты, заданные для элемента `<behavior>`, чтобы передать данные библиотеке. После перезапуска приложения наш клиент получает новую функциональность, которую можно немедленно использовать, обращаясь к действиям из шаблонов HXML.

Добавление сообщений

Действия `phone` и `email`, добавленные в предыдущем разделе, являются примерами «системных действий». Системные действия инициируют пользовательский интерфейс или функциональность, предоставляемую ОС на устройстве. Но нестандартные действия не ограничиваются взаимодействием с API уровня ОС. Напомним, что обратные вызовы, реализующие действия, могут выполнять произвольный код, в том числе код, который рендерит собственные UI-элементы. Следующий пример нестандартного действия делает именно это: он рендерит UI-элемент для нестандартного подтверждающего сообщения.

Как уже отмечалось, веб-приложение Contacts выводит сообщения об успешных действиях, например удалении или создании контакта. Эти сообщения генерируются бэкендом Flask с использованием функции `flash()`, вызываемой из представлений. Затем базовый шаблон layout.html рендерит сообщения на финальную веб-страницу.

Листинг 222. Фрагмент templates/layout.html

```
{% for message in get_flashed_messages() %}
 <div class="flash">{{ message }}</div>
{% endfor %}
```

Приложение Flask все еще включает вызовы `flash()`, однако приложение Hyperview не обращается к сообщению, которое должно выводиться для пользователя. Добавим эту поддержку.

Для вывода сообщений можно поступить так же, как в веб-приложении: перебрать сообщения в цикле и отрендерить элементы `<text>` в layout.xml. У этого подхода был серьезный недостаток: отрендеренные сообщения привязывались к конкретному экрану. Если экран скрыт навигационным действием, сообщение тоже будет скрыто. Но нам нужно, чтобы интерфейс сообщения отображался «над» всеми экранами в стеке навигации. В этом случае сообщение останется видимым (и исчезнет через несколько секунд) даже при изменении находяще-

гося под ним стека экранов. Чтобы часть пользовательского интерфейса отображалась вне элементов `<screen>`, придется расширить клиент Hyperview новым нестандартным действием `show-message`. Для этого снова воспользуемся библиотекой с открытым исходным кодом `react-native-root-toast`, которую нужно добавить в приложение.

```
> cd hyperview/demo
> yarn add react-native-root-toast ❶
> yarn start ❷
```

❶ Добавляет зависимость от react-native-root-toast.
❷ Перезапускает мобильное приложение.

Теперь можно написать код для реализации UI сообщений как нестандартного действия.

Листинг 223. demo/src/message.js

```
import Toast from 'react-native-root-toast'; ❶

const namespace = "https://hypermedia.systems/hyperview/message";

export default {
 action: "show-message", ❷
 callback: (behaviorElement) => { ❸
  const text = behaviorElement.getAttributeNS(namespace, "text");
  if (text != null) {
   Toast.show(text, {position: Toast.positions.TOP, duration: 2000}); ❹
  }
 },
};
```

❶ Импортирует Toast API.
❷ Имя действия.
❸ Обратный вызов, выполняемый при срабатывании действия.
❹ Передает сообщение библиотеке toast.

Код очень похож на реализацию `open-phone`. Оба обратных вызова строятся по одной схеме: прочитать атрибуты с уточнением пространства имен из элемента `<behavior>` и передать их значения сторонней библиотеке. Для простоты мы жестко запрограммируем настройки для вывода сообщения в верхней части экрана и удаления через 2 секунды. Однако библиотека `react-native-root-toast` предоставляет многочисленные возможности для позиционирования, настройки хронометража анимаций, цветов и многого другого. Эти параметры задаются включением дополнительных атрибутов в `behaviorElement`. Для текущей задачи мы ограничимся минимальной реализацией.

Теперь необходимо зарегистрировать нестандартное действие в компоненте `<Hyperview>`, передав его свойству `behaviors`.

Листинг 224. demo/src/HyperviewScreen.js

```
import React, { PureComponent } from 'react';
import Hyperview from 'hyperview';
import OpenEmail from './email';
import OpenPhone from './phone';
import ShowMessage from './message'; ❶

export default class HyperviewScreen extends PureComponent {
 // Фрагмент опущен для краткости

 behaviors = [OpenEmail, OpenPhone, ShowMessage]; ❷

 // Фрагмент опущен для краткости
}
```

❶ Импортировать действие show-message.
❷ Передать действие компоненту Hyperview в свойстве с именем behaviors.

Последнее, что остается сделать, — инициировать действие `show-message` из разметки HXML. Сообщение может выводиться в результате трех действий пользователя:

1) создания нового контакта;
2) обновления существующего контакта;
3) удаления контакта.

Первые два действия реализуются в приложении с использованием того же шаблона HXML, form_fields.xml. При успешном создании или обновлении контакта шаблон перезагружает экран и инициирует событие, используя варианты поведения, срабатывающие по триггеру `load`. Действие удаления также использует варианты поведения, срабатывающие по триггеру `load`, определенные в шаблоне deleted.xml. Таким образом, шаблоны form_fields.xml и deleted.xml необходимо изменить, чтобы они также выводили сообщения при загрузке. Так как фактическое поведение в обоих шаблонах будет одинаковым, создадим общий шаблон для повторного использования HXML.

Листинг 225. hv/templates/messages.xml

```
{% for message in get_flashed_messages() %}
 <behavior ❶
  xmlns:message="https://hypermedia.systems/hyperview/message"
  trigger="load" ❷
  action="show-message" ❸
```

```
  message:text="{{ message }}" ❹
 />
{% endfor %}
```

❶ Определяет поведение для отображения всех сообщений.
❷ Поведение инициируется непосредственно при загрузке элемента.
❸ Инициирует новое действие show-message.
❹ Действие show-message выводит флеш-сообщение.

Как и в файле layout.html веб-приложения, мы перебираем все флеш-сообщения и рендерим фрагмент разметки для каждого сообщения. Однако в веб-приложении сообщение рендерилось напрямую на странице. В приложении Hyperview каждое сообщение выводится с использованием поведения, инициирующего нестандартный пользовательский интерфейс. Остается лишь добавить этот шаблон в form_fields.xml.

Листинг 226. Фрагмент hv/templates/form_fields.xml

```
<view xmlns="https://hyperview.org/hyperview" style="edit-group">
 {% if saved %}
  {% include "hv/messages.xml" %} ❶
  <behavior trigger="load" once="true" action="dispatch-event" event-
name="contact-updated" />
  <behavior trigger="load" once="true" action="reload"
href="/contacts/{{contact.id}}" />
 {% endif %}
 <!-- Фрагмент опущен для краткости -->
</view>
```

❶ Выводит сообщения непосредственно при загрузке экрана.

То же самое можно сделать в deleted.xml.

Листинг 227. hv/templates/deleted.xml

```
<view xmlns="https://hyperview.org/hyperview">
 {% include "hv/messages.xml" %} ❶
 <behavior trigger="load" action="dispatch-event" event-name="contact-
updated" />
 <behavior trigger="load" action="back" />
</view>
```

❶ Выводит сообщения непосредственно при загрузке экрана.

Как в form_fields.xml, так и в deleted.xml по триггеру `load` инициируются сразу несколько поведений. В deleted.xml происходит немедленный возврат к предыдущему экрану. В form_fields.xml текущий экран немедленно перезагружается

для вывода подробной информации контакта. Если бы элементы UI сообщения рендерились прямо на экране, пользователь вряд ли успел бы увидеть сообщение, прежде чем экран исчезнет или будет перезагружен. Благодаря использованию нестандартного действия интерфейс сообщения остается видимым даже при изменении находящихся под ним экранов.

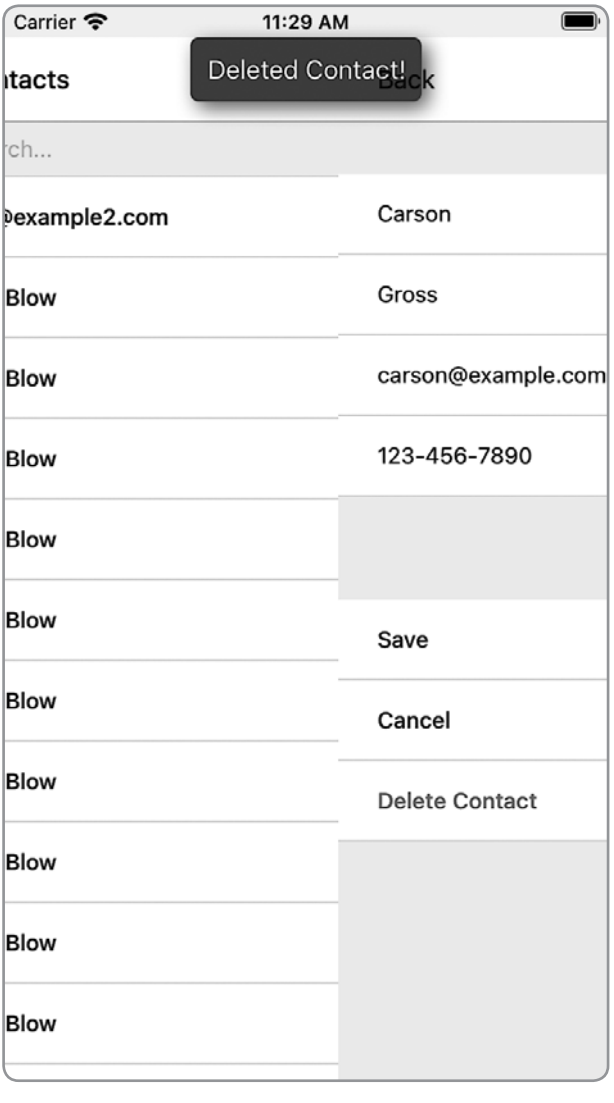


Рис. 20. Сообщение, выводимое при возврате к предыдущему экрану

Жест смахивания

Чтобы добавить поддержку коммуникаций и пользовательский интерфейс сообщения, мы расширили клиент нестандартными действиями поведения. Однако клиент Hyperview можно расширить и нестандартными компонентами UI, которые рендерятся на экране. Нестандартные компоненты реализуются в виде компонентов React Native. А это означает, что все, что делается в React Native, можно сделать и в Hyperview! Нестандартные компоненты открывают бесконечные возможности для создания полнофункциональных мобильных приложений на основе архитектуры Hypermedia.

Чтобы продемонстрировать эти возможности, расширим клиент Hyperview в мобильном приложении и добавим в него компонент «строка с поддержкой смахивания». Как он работает? Такой компонент поддерживает жест горизонтального смахивания (свайпа). Когда пользователь проводит по компоненту справа налево, компонент смещается в сторону, открывая набор кнопок действий. При нажатии этих кнопок может инициироваться стандартное поведение Hyperview. Мы воспользуемся этим нестандартным компонентом на экране со списком контактов. Каждый контакт будет представлять собой «строку с поддержкой смахивания», а действия предоставят быстрый доступ к действиям редактирования и удаления для контакта.

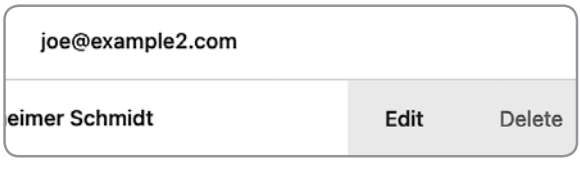


Рис. 21. Элемент списка с поддержкой смахивания

Проектирование компонента

Чтобы не реализовывать жест смахивания с нуля, снова воспользуемся сторонней библиотекой с открытым исходным кодом: `react-native-swipeable`.

```
> cd hyperview/demo
> yarn add react-native-swipeable ❶
> yarn start ❷
```

❶ Добавляет зависимость от react-native-swipeable.
❷ Перезапускает мобильное приложение.

Эта библиотека предоставляет компонент React Native с именем `Swipeable`. Она может рендерить любые компоненты React Native как свой основной контент

(часть интерфейса, к которой может применяться жест смахивания). Она также получает массив компонентов React Native как свойство, которое должно рендериться как кнопки действий.

При проектировании нестандартного компонента определять HXML компонента лучше перед написанием кода. Это позволяет создать разметку, которая будет выразительной, но компактной и будет работать с используемой библиотекой.

Для реализации строки с поддержкой смахивания нам понадобится способ представления всего компонента, основного контента и отдельной кнопки.

```
<swipe:row xmlns:swipe="https://hypermedia.systems/hyperview/swipeable"> ❶
 <swipe:main> ❷
  <!-- основной контент -->
 </swipe:main>

 <swipe:button> ❸
  <!-- первая кнопка, которая появляется при смахивании -->
 </swipe:button>

 <swipe:button> ❹
  <!-- вторая кнопка, которая появляется при смахивании -->
 </swipe:button>
</swipe:row>
```

❶ Родительский элемент инкапсулирует всю строку с поддержкой смахивания, для него определяется нестандартное пространство имен.

❷ Главный контент строки с поддержкой смахивания; может содержать произвольную разметку HXML.

❸ Первая кнопка, которая появляется при смахивании; может содержать произвольную разметку HXML.

❹ Вторая кнопка, которая появляется при смахивании; может содержать произвольную разметку HXML.

Подобная структура четко отделяет основной контент от кнопок. Она также поддерживает одну, две или несколько кнопок. Кнопки выводятся в порядке определения, что позволяет легко менять их местами при необходимости.

Эта структура включает все, что необходимо для реализации строки с поддержкой смахивания для списка контактов. При этом она создана в достаточно обобщенном виде, так что может использоваться повторно. Предыдущая разметка никак не связана с именем контакта, редактированием контакта или его удалением. Если позднее в приложение будет добавлен другой экран со списком, можно будет воспользоваться этим компонентом, чтобы реализовать поддержку смахивания для пунктов этого списка.

Реализация компонента

Итак, теперь вы знаете структуру HXML нестандартного компонента, и можно написать его код. Как он будет выглядеть? Компоненты Hyperview пишутся как компоненты React Native. Такие компоненты React Native связываются с уникальным пространством имен XML и именем тега. Когда клиент Hyperview встречает это пространство имен и имя тега в HXML, он делегирует рендеринг элемента HXML соответствующему компоненту React Native. В одной из фаз делегирования клиент Hyperview передает несколько свойств компоненту React Native:

- `element`: элемент XML DOM, соответствующий компоненту React Native;
- `stylesheets`: стили, определенные в `<screen>`;
- `onUpdate`: функция, вызываемая при инициировании поведения компонентом;
- `option`: различные настройки, используемые клиентом Hyperview.

Наш компонент представляет собой контейнер со слотами, в которых рендерится произвольный основной контент и кнопки. Это означает, что он должен делегировать рендеринг этих частей пользовательского интерфейса клиенту Hyperview. Для этого используется общедоступная функция, предоставляемая клиентом Hyperview, `Hyperview.renderChildren()`.

Теперь вы знаете, как реализуются нестандартные компоненты Hyperview, и мы можем написать код строки с поддержкой смахивания.

Листинг 228. demo/src/swipeable.js

```
import React, { PureComponent } from 'react';
import Hyperview from 'hyperview';
import Swipeable from 'react-native-swipeable';

const NAMESPACE_URI = 'https://hypermedia.systems/hyperview/swipeable';

export default class SwipeableRow extends PureComponent { ❶
 static namespaceURI = NAMESPACE_URI; ❷
 static localName = "row"; ❸

 getElements = (tagName) => {
  return
Array.from(this.props.element.getElementsByTagNameNS(NAMESPACE_URI,
tagName));
 };

 getButtons = () => { ❹
  return this.getElements("button").map((buttonElement) => {
   return Hyperview.renderChildren(buttonElement, this.props.stylesheets,
```

```
this.props.onUpdate, this.props.options); ❺
  });
 };

 render() {
  const [main] = this.getElements("main");
  if (!main) {
   return null;
  }

  return (
   <Swipeable rightButtons={this.getButtons()}> ❻
    {Hyperview.renderChildren(main, this.props.stylesheets,
this.props.onUpdate, this.props.options)} ❼
   </Swipeable>
  );
 }
}
```

❶ Компонент React Native на основе класса.
❷ Связывает компонент с пространством имен HXML.
❸ Связывает компонент с именем тега HXML.
❹ Функция возвращает массив компонентов React Native для каждого элемента <button>.
❺ Делегирует рендеринг каждой кнопки клиенту Hyperview.
❻ Кнопки и основной контент передаются сторонней библиотеке.
❼ Делегирует клиенту Hyperview рендеринг основного контента.

Класс `SwipeableRow` реализует компонент React Native. В начале класса задаются значения статических свойств `namespaceURI` и `localName`. Эти свойства связывают компонент React Native с уникальной парой пространства имен и имени тега в HXML. Так клиент Hyperview узнает, что при обнаружении нестандартных элементов в HXML следует делегировать операцию `SwipeableRow`.

В конце класса находится метод `render()`. `render()` вызывается React Native для возвращения отрендеренного компонента. Так как React Native строится по принципу композиции, `render()` обычно возвращает композицию других компонентов React Native. В данном случае возвращается компонент `Swipeable` (предоставляемый библиотекой `react-native-swipeable`), включающий компоненты React Native для кнопок и основного контента. Компоненты React Native для кнопок и основного контента создаются аналогично.

- Найти конкретные дочерние элементы (`<button>` или `<main>`).
- Преобразовать эти элементы в компоненты React Native при помощи `Hyperview.renderChildren()`.
- Назначить компоненты как дочерние элементы или свойства `Swipeable`.

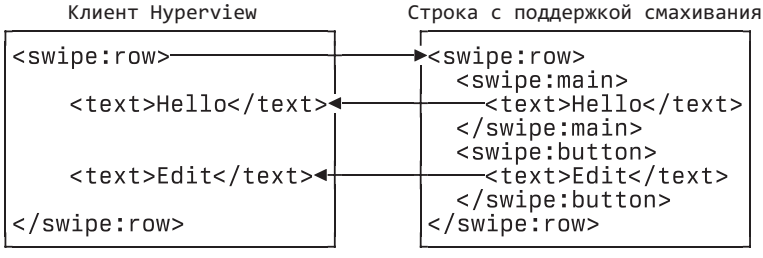


Рис. 22. Делегирование рендеринга между клиентом и нестандартными компонентами

Если вы никогда не работали с React или React Native, код может показаться вам непонятным. Ничего страшного. Здесь важен вывод: можно написать код для преобразования произвольной разметки HXML в компоненты React Native. Структура HXML (атрибуты и элементы) может использоваться для представления разных частей UI (в данном случае кнопок и основного контента). Наконец, код может делегировать рендеринг дочерних компонентов обратно клиенту Hyperview.

Результат: компонент строки с поддержкой смахивания полностью обобщен. Фактическая структура, стили и взаимодействия основного контента и кнопок могут определяться в HXML. Создание обобщенного компонента означает, что этот компонент можно будет использовать повторно между разными экранами для различных целей. Если в дальнейшем будут добавлены новые нестандартные компоненты или новые действия поведения, они будут работать с нашей реализацией строк с поддержкой смахивания.

Последнее, что осталось, — зарегистрировать новый компонент в клиенте Hyperview. Процесс похож на регистрацию нестандартных действий. Нестандартные компоненты передаются компоненту Hyperview в отдельном свойстве `components`.

Листинг 229. demo/src/HyperviewScreen.js

```
import React, { PureComponent } from 'react';
import Hyperview from 'hyperview';
import OpenEmail from './email';
import OpenPhone from './phone';
import ShowMessage from './message';
import SwipeableRow from './swipeable'; ❶

export default class HyperviewScreen extends PureComponent {
 // ...

 behaviors = [OpenEmail, OpenPhone, ShowMessage];
 components = [SwipeableRow]; ❷
```

```
 render() {
  return (
   <Hyperview
    behaviors={this.behaviors}
    components={this.components} ❸
    entrypointUrl={this.entrypointUrl}
    // Другие свойства...
   />
  );
 }
}
```

❶ Импортирует компонент SwipeableRow.
❷ Создает массив нестандартных компонентов.
❸ Передает нестандартный компонент компоненту Hyperview в свойстве с именем components.

Теперь все готово к обновлению шаблонов HXML, чтобы использовать новый компонент строки с поддержкой смахивания.

Использование компонента

Сейчас разметка HXML для контакта в списке состоит из элементов `<behavior>` и `<text>`.

Листинг 230. Фрагмент hv/rows.xml

```
<item key="{{ contact.id }}" style="contact-item">
 <behavior trigger="press" action="push" href="/contacts/{{ contact.id }}"
/> ❶
 <text style="contact-item-label">
  <!—Фрагмент опущен длякраткости -->
 </text>
</item>
```

С нашим компонентом эта разметка становится основным пользовательским интерфейсом. Начнем с добавления `<row>` и `<main>` как родительских элементов.

Листинг 231. Добавление строки с поддержкой смахивания hv/rows.xml

```
<item key="{{ contact.id }}">
 <swipe:row xmlns:swipe="https://hypermedia.systems/hyperview/swipeable"> ❶
  <swipe:main> ❷
   <view style="contact-item"> ❸
    <behavior trigger="press" action="push" href="/contacts/{{ contact.id
}}" /> ❶
    <text style="contact-item-label">
     <!—Фрагмент опущен для краткости -->
    </text>
```

```
   </view>
  </swipe:main>
 </swipe:row>
</item>
```

❶ Добавляет родительский элемент <swipe:row> с псевдонимом пространства имен для swipe.
❷ Добавляет элемент <swipe:main> для определения основного контента.
❸ Упаковывает существующие элементы <behavior> и <text> в элемент <view>.

Ранее стиль `contact-item` назначался элементу `<item>`. Это имело смысл, когда элемент `<item>` был контейнером для основного контента элемента списка. Теперь, когда основной контент является дочерним элементом `<swipe:main>`, необходимо ввести новый элемент `<view>` для применения стилей.

Если перезагрузить бэкенд и мобильное приложение, на экране со списком контактов не будет заметно никаких изменений. Пока не определены кнопки действий, ничего не открывается при смахивании на строке. Добавим две кнопки в компонент.

Листинг 232. Добавление кнопок в hv/rows.xml

```
<item key="{{ contact.id }}">
 <swipe:row xmlns:swipe="https://hypermedia.systems/hyperview/swipeable"> ❶
  <swipe:main>
   <!—Фрагмент опущен для краткости  -->
  </swipe:main>

  <swipe:button> ❶
   <view style="swipe-button">
    <text style="button-label">Edit</text>
   </view>
  </swipe:button>

  <swipe:button> ❷
   <view style="swipe-button">
    <text style="button-label-delete">Delete</text>
   </view>
  </swipe:button>
 </swipe:row>
</item>
```

❶ Добавляет <swipe:button> для действия редактирования.
❷ Добавляет <swipe:button> для действия удаления.

Если теперь запустить мобильное приложение, мы увидим, что строка с поддержкой смахивания работает! Если свайпнуть по пункту списка, на экране появляются кнопки Edit (Редактировать) и Delete (Удалить). Правда, пока они

еще ничего не делают. К этим кнопкам необходимо добавить поведение. С кнопкой Edit все просто: при нажатии должен открываться экран с подробной информацией о контакте в режиме редактирования.

Листинг 233. Фрагмент hv/rows.xml

```
<swipe:button>
 <view style="swipe-button">
  <behavior trigger="press" action="push" href="/contacts/{{ contact.id
}}/edit" /> ❶
  <text style="button-label">Edit</text>
 </view>
</swipe:button>
```

❶ При нажатии в стек проталкивается новый экран с интерфейсом редактирования.

С кнопкой Delete сложнее. Экрана, который можно было бы открыть для удаления, не существует, — так что же должно происходить при нажатии этой кнопки? Вероятно, стоит использовать такое же взаимодействие, что и у кнопки Delete на экране редактирования контакта. Это взаимодействие открывает системное диалоговое окно, которое предлагает пользователю подтвердить удаление. Если пользователь подтверждает операцию, клиент Hyperview выдает запрос `POST` к `/contacts/<contact_id>/delete` и присоединяет ответ к экрану. Ответ немедленно инициирует поведение для перезагрузки списка контактов и вывода сообщения. Это взаимодействие подойдет и для нашей кнопки действия.

Листинг 234. Фрагмент hv/rows.xml

```
<swipe:button>
 <view style="swipe-button">
  <behavior ❶
   xmlns:alert="https://hyperview.org/hyperview-alert"
   trigger="press"
   action="alert"
   alert:title="Confirm delete"
   alert:message="Are you sure you want to delete {{ contact.first }}?"
  >
   <alert:option alert:label="Confirm">
    <behavior ❷
     trigger="press"
     action="append"
     target="item-{{ contact.id }}"
     href="/contacts/{{ contact.id }}/delete"
     verb="post"
    />
   </alert:option>
   <alert:option alert:label="Cancel" />
  </behavior>
  <text style="button-label-delete">Delete</text>
```

```
 </view>
</swipe:button>
```

❶ При нажатии открывает системное диалоговое окно, в котором пользователю предлагается подтвердить действие («Вы уверены, что хотите удалить...?»).

❷ Если пользователь подтверждает действие, выдает запрос POST к конечной точке удаления и присоединяет ответ к родительскому элементу <item>.

Теперь при нажатии Delete появляется диалоговое окно для подтверждения, как и ожидалось. После подтверждения ответ бэкенда инициирует поведение, которое выводит подтверждающее сообщение и перезагружает список контактов. Удаленный контакт исчезает из списка.

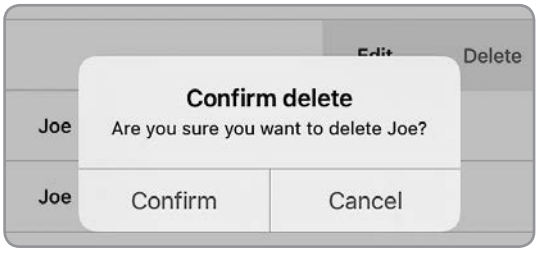


Рис. 23. Удаление с использованием кнопки, открываемой смахиванием

Заметим, что кнопки действий могут поддерживать любые типы действий поведения, от `push` до `alert`. При желании можно сделать так, чтобы кнопки действий инициировали нестандартные действия, такие как `open-phone` и `open-email`. Нестандартные компоненты и действия можно свободно смешивать со стандартными компонентами и действиями фреймворка Hyperview. В результате расширения клиента Hyperview работают как встроенные функции.

Поделимся с вами одним секретом. Внутри клиента Hyperview стандартные компоненты и действия реализуются точно так же, как нестандартные компоненты и действия! Код рендеринга работает с `<view>` точно так же, как с `<swipe:row>`. Код поведения работает с `alert` так же, как с `open-phone`. Оба реализуются средствами, описанными в этом разделе. Стандартные компоненты и действия отличаются только тем, что они нужны во всех мобильных приложениях. Однако это всего лишь отправная точка для дальнейших расширений.

Многим мобильным приложениям требуются расширения клиента Hyperview, чтобы обеспечивать качественное взаимодействие с пользователем. Расширения превращают обобщенный «клиент Hyperview» в специализированный клиент, созданный для нужд конкретного приложения. И что очень важно, при этом сохраняются гипермедийная, управляемая сервером архитектура и все ее преимущества.

Мобильные гипермедиа-управляемые приложения

На этом работа над мобильным приложением Contact.app подходит к концу. Отойдем от подробностей кода и рассмотрим общие принципы.

- Основная логика приложения размещается на сервере.
- Шаблоны, которые рендерятся сервером, обеспечивают работу как веб-, так и мобильных приложений.
- Нативная специализация достигается использованием скриптов в веб-приложениях и настройкой клиента в мобильных приложениях.

Архитектура гипермедиа-управляемых приложений обеспечила широкие возможности повторного использования кода и управления технологическим стеком. Текущее обновление и обслуживание веб- и мобильных приложений можно проводить одновременно.

Да, у мобильных гипермедиа-управляемых приложений хорошие перспективы.

Заметки о гипермедиа: достаточно хороший интерфейс и островки интерактивности

При переходе на методологию HDA многие разработчики приложений SPA и нативных мобильных приложений сталкиваются с проблемой: они смотрят на свое текущее приложение и пытаются представить его реализацию на основе гипермедиа. Хотя `htmx` и Hyperview значительно улучшают качество взаимодействия с пользователем благодаря гипермедиа-управляемому подходу, иногда реализовать конкретный опыт взаимодействия оказывается достаточно сложно.

Как было показано в главе 2, Рой Филдинг упоминал об этом компромиссе в контексте RESTful-архитектуры, где «...информация передается в стандартизированной форме, а не в той, которая адаптирована для потребностей приложения».

Реализация чуть менее эффективного и интерактивного решения для конкретного UX избавит вас от лишней сложности при построении приложений.

Лучшее не должно быть врагом хорошего. Иногда предоставление пользователю чуть менее сложного опыта взаимодействия приносит множество преимуществ, и правильное применение таких инструментов, как `htmx` и Hyperview, делает этот компромисс намного более привлекательным.

Заключение

Переосмысление гипермедиа

Надеемся, мы убедили вас, что не стоит думать о гипермедиа как об «устаревшей» технологии или технологии, подходящей только для создания «документов» со ссылками, текстом и графикой. По сути, это эффективная технология для построения *приложений*. Из этой книги вы узнали, как строить сложные пользовательские интерфейсы (для веб-приложений на основе `htmx` и мобильных приложений на основе Hyperview) с использованием гипермедиа как базовой технологии.

Многие веб-разработчики считают ссылки и формы «простого» HTML устаревшими инструментами менее требовательной эпохи. И в каком-то отношении они правы: ранней веб-среде определенно не хватало удобства использования. Однако сейчас существуют библиотеки JavaScript, расширяющие HTML и избавляющие от его основных ограничений. Например, с `htmx` можно:

- наделить любой элемент возможностью выдавать запрос HTTP;
- наделить любое событие возможностью инициировать событие HTTP;
- использовать все доступные типы методов HTTP;
- выбрать любой элемент DOM целевым для замены.

Все это позволило нам построить для Contact.app пользовательские интерфейсы, которые, по мнению многих разработчиков, требуют большого объема клиентского кода JavaScript, а мы использовали концепции гипермедиа.

Гипермедиа-управляемая архитектура подходит не для каждого приложения. Однако для многих приложений повышенная гибкость и простота гипермедиа могут стать огромным преимуществом. Даже если этот подход не улучшит ваше приложение, вам стоит понять его суть, сильные и слабые стороны и отличия от применяемой вами методики. Веб-среда росла быстрее, чем любая другая распределенная система в истории, и веб-разработчики должны уметь использовать сильные стороны базовых технологий, которые сделали возможным этот рост.

Остановитесь и задумайтесь

Сообщество JavaScript — а если на то пошло, то и все сообщество веб-разработчиков — известно своей непредсказуемостью. Новые фреймворки и техно-

логии появляются каждый месяц, а иногда и *каждую неделю*. Отслеживание новейших и лучших технологий отнимает много сил, но в то же время мы приходим в ужас от перспективы отстать от трендов и погубить тем самым свою карьеру.

Этот страх не лишен оснований; есть немало опытных разработчиков, чья карьера застопорилась из-за того, что технология, на которой они решили специализироваться, оказалась нежизнеспособной. В веб-разработке многие компании предпочтут молодого специалиста вместо опытного разработчика, который «отстал от жизни».

Таковы реалии нашей отрасли, и о них нельзя умалчивать. Вместе с тем не следует игнорировать проблемы, которые эти реалии создают. Это среда высокого давления, где все постоянно ждут новейшей и самой мощной технологии, которая вдруг все изменит. Это заставляет разработчиков утверждать, что именно *их* технология такова. Это заставляет отдавать предпочтение *сложности* перед *простотой*. Люди боятся спросить: «А не слишком ли это сложно?», потому что это будет звучать как «Мне не хватает мозгов, чтобы это понять».

В разработке ПО (и особенно в веб-разработке) существует тенденция гнаться за инновациями вместо того, чтобы глубже разбираться в существующих технологиях и развивать их. Мы предпочитаем искать новые гениальные решения вместо того, чтобы обращаться к проверенным идеям. Это можно понять: технологическая отрасль обречена смотреть в будущее.

С другой стороны, как вы поняли из определения REST Роя Филдинга, первые веб-архитекторы иногда предлагали замечательные идеи, которые не получили должного внимания. Мы живем так долго, что стали свидетелями, как гипермедиа возвращается в качестве «хорошо забытого старого». Удивительно, что такая эффективная идея, как REST, была так легко забыта в отрасли. К счастью, концепции никуда не исчезают, ожидая, пока мы снова откроем их и вдохнем в них жизнь. Исходная RESTful-архитектура веб-среды, если взглянуть на нее по-новому, способна решить многие проблемы современной веб-разработки.

Возможно, по совету Марка Твена, настало время остановиться и задуматься. Может быть, в эту минуту спокойствия мы сможем выйти из бесконечного круговорота «инноваций», вернуться к истокам веб-технологий и заняться их изучением.

Возможно, пришло время дать гипермедиа шанс.

Карсон Гросс, Адам Степински, Денис Акшимшек

Hypermedia-разработка. htmx и Hyperview

Перевел с английского Е. Матвеев

Руководитель дивизиона	*Ю. Сергиенко*
Ведущий редактор	*Е. Строганова*
Технический редактор	*Н. Хлебина*
Литературный редактор	*М. Трусковская*
Художественный редактор	*В. Мостипан*
Корректоры	*Н. Викторова, Т. Никифорова*
Верстка	*Л. Егорова*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 08.2024. Наименование: книжная продукция. Срок годности: не ограничен.

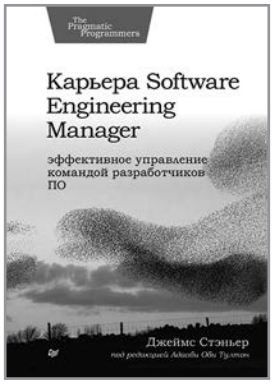
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 19.06.24. Формат 70×100/16. Бумага офсетная. Усл. п. л. 29,670. Тираж 700. Заказ 0000.

Джеймс Стэньер

КАРЬЕРА SOFTWARE ENGINEERING MANAGER. ЭФФЕКТИВНОЕ УПРАВЛЕНИЕ КОМАНДОЙ РАЗРАБОТЧИКОВ ПО



Перед вами неожиданно открылась возможность возглавить команду разработчиков ПО? Пора становиться менеджером? Как решить, подходит ли вам такой шаг в карьере? И если да, то чему нужно научиться, чтобы добиться успеха? С чего начать? Как понять, что вы все делаете правильно? Что вообще означает «менеджмент»? Джеймс Стэньер делится секретами, которые необходимо знать, чтобы успешно управлять командой разработчиков.

Смена статуса с «инженер-разработчик» на «руководитель команды» не должна вас пугать — инженеры могут быть менеджерами, причем идеальными.

Отбросьте болтовню и сосредоточьтесь на практических методах и инструментах. Вы станете эффективным лидером команды, на которого будут равняться ваши сотрудники.

Великие менеджеры могут сделать мир лучше. Присоединяйтесь к нам!

Робин Никсон

СОЗДАЕМ ДИНАМИЧЕСКИЕ ВЕБ-САЙТЫ С ПОМОЩЬЮ PHP, MYSQL, JAVASCRIPT, CSS И HTML5

6-е издание

Новое издание бестселлера описывает как клиентские, так и серверные аспекты веб-разработки. Книга, наполненная ценными практическими советами и подробным теоретическим материалом, поможет вам освоить динамическое веб-программирование с применением самых современных технологий. Для закрепления усвоенных знаний автор расскажет, как создать полнофункциональный сайт, работающий по принципу социальной сети.

Билл Любанович

FASTAPI: ВЕБ-РАЗРАБОТКА НА PYTHON

FastAPI — относительно новый, но надежный фреймворк с чистым дизайном, использующий преимущества актуальных возможностей Python. Как следует из названия, FastAPI отличается высоким быстродействием и способен конкурировать в этом с аналогичными фреймворками на таких языках, как Golang. Эта практическая книга расскажет разработчикам, знакомым с Python, как FastAPI позволяет достичь большего за меньшее время и с меньшим количеством кода.

Билл Любанович рассказывает о тонкостях разработки с применением FastAPI и предлагает множество рекомендаций по таким темам, как формы, доступ к базам данных, графика, карты и многое другое, что поможет освоить основы и даже пойти дальше. Кроме того, вы познакомитесь с RESTful API, приемами валидации данных, авторизации и повышения производительности. Благодаря сходству с такими фреймворками, как Flask и Django, вы легко начнете работу с FastAPI.