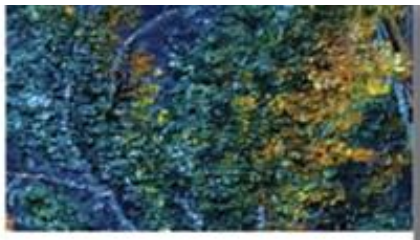

INTERNATIONAL SCIENCE REVIEWS

AIU

№ 2 (3) 2022

Natural Sciences and Technologies series

www.aiu.kz

ISSN: 2707-4862

ASTANA INTERNATIONAL UNIVERSITY

INTERNATIONAL SCIENCE REVIEWS
Natural Sciences and Technologies series

Has been published since 2020

№2 (3) 2022

Nur-Sultan

Editorial address: 8, Kabanbay Batyr avenue, of.316, Nur-Sultan,
Kazakhstan, 010000
Tel.: (7172) 24-18-52 (ext. 316)
E-mail: natural-sciences@aiu.kz

International Sciense Reviews NST - 76153
International Science Reviews
Natural Sciences and Technologies series
Owner: Astana International University
Periodicity: quarterly
Circulation: 500 copies

CONTENT

ПРЕИМУЩЕСТВА КЛАССИЧЕСКИХ АРХИТЕКТУРНЫХ РЕШЕНИЙ ДЛЯ ВЕБ-ПРИЛОЖЕНИЙ.

Кабиев Аслан, Нагим Шайдолла

Магистранты 2 курса Международного университета Астана, г.Нур-Султан

Аннотация. ***Предмет.*** Современное веб-приложение, очевидно, реализовано на JavaScript и будет генерировать свой HTML-код на клиенте в браузере. Он взаимодействует с сервером только путем получения данных в формате JSON из конечной точки HTTP/REST API — это, кажется, общепринятое мнение на сегодняшний день. Но действительно ли серверный HTML и прогрессивное улучшение устарели? Наоборот, мы можем использовать эти технологии для создания приложений, которые часто намного лучше, чем те, которые мы могли бы создать с помощью Framework of the Week для создания одностраничных приложений.

Цели. Изучение преимущества классических архитектурных решений для веб-приложений.

Методология. В процессе иследования использовались методы логического, статического анализа

Результаты. На практике мы убедились, что когда мы разрабатываем приложения с серверным рендерингом HTML и пользовательскими элементами, тщательно модульным CSS и небольшим количеством JavaScript, мы можем создавать архитектурно чистые, компактные, быстрые и эргономичные приложения. Мы не боимся сравнения этих приложений с приложениями, созданными с использованием популярных подходов SPA — во многих аспектах они часто превосходят их. Мы обнаружили, что они не только имеют меньшие коммуникационные издержки, более высокую отказоустойчивость и более легко доступны, но и их легче поддерживать в долгосрочной перспективе.

Выводы. Мы зависим от браузера и его стандартов, а не от фреймворка SPA, который может полностью измениться в следующей версии или может быть заменен следующей альтернативной фреймворком.

Ключевые слова: архитектура, веб-приложения, CSS, HTML, сервер

Давайте рассмотрим проблемы, которые часто возникают при использовании одностраничных приложений (SPA). При традиционном разделении между сервером и клиентом клиент отвечает за внешний вид и поведение. Состояние, бизнес-логика, маршрутизация, логика представления и шаблоны находятся исключительно на сервере (рис. 1).

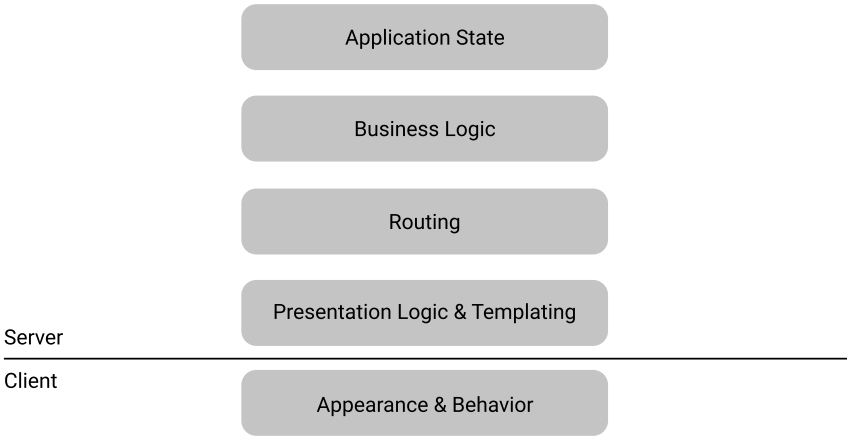

Рисунок 1. Традиционное разделение между сервером и клиентом

Когда мы создаем приложение и хотим оптимизировать его, чтобы оно быстрее реагировало и, возможно, предоставляло автономные возможности, мы часто переносим некоторые обязанности с сервера на клиента. Многие компании считают дополнительным преимуществом, если мы можем разделить эти обязанности между разными отделами. Тогда экспертам по бэкенду нужно только предоставить API, но им не нужно заботиться о HTML/CSS/JS. Кроме того, существует распространенное заблуждение, что JSON меньше, чем HTML. Джон Мур объясняет в статье , почему это не так. Часто логика представления и шаблоны перемещаются во внешний интерфейс. Это относится к таким технологиям, как React или Vue.js (рис. 2).

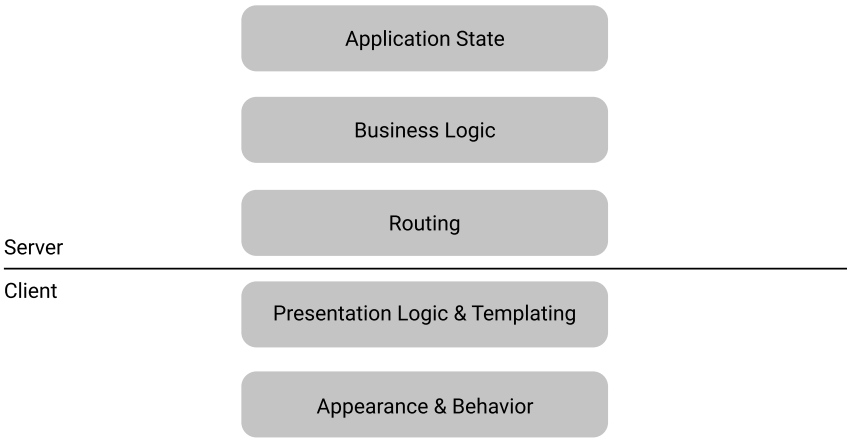

Рисунок 2. Логика представления и шаблоны перенесены во внешний интерфейс

Следующим шагом часто является передача маршрутизации клиенту. Например, это происходит в React с React Router и в Vue.js с Vue Router. В таких фреймворках, как Angular или Ember, клиент также отвечает за маршрутизацию (рис. 3).

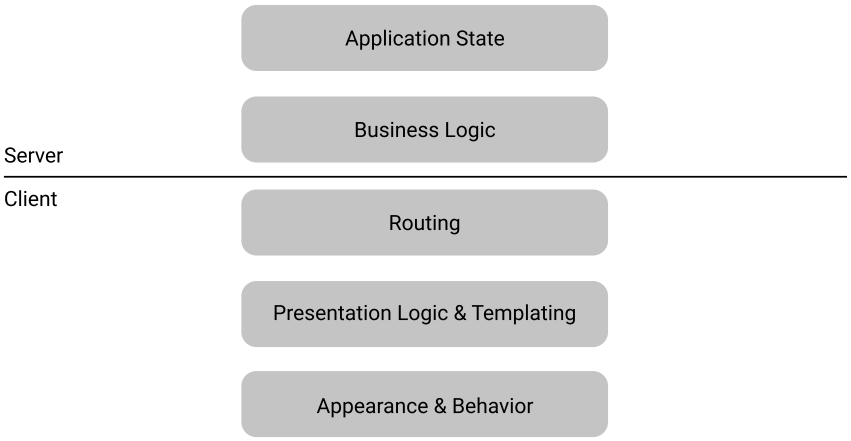

Рисунок 3. Маршрутизация перемещается в клиент

Может показаться, что перекладывание обязанностей с сервера на клиента — это всего лишь смена границ. Однако это изменение границы имеет дополнительные последствия, которые нам необходимо рассмотреть.

Раньше мы только интерпретировали HTML в клиенте — функциональность, которую браузер предоставляет из коробки. Теперь нам нужен клиент JSON и API, которые мы можем использовать. Это также добавляет уровень косвенности: в логике представления у нас больше нет прямого доступа к бизнес-логике и состоянию приложения, но вместо этого мы должны преобразовывать данные в JSON и обратно. Определение и изменение этого API требует высокого уровня координации и взаимодействия, особенно когда сервер и клиент разрабатываются разными командами.

Если и бизнес-логика, и состояние приложения полностью остаются на сервере, у нас либо такое же, либо больше взаимодействие между клиентом и сервером. Однако если мы переместим часть бизнес-логики на клиент, нам теперь нужно будет постоянно решать, какая бизнес-логика принадлежит серверу, какая бизнес-логике принадлежит клиенту, а какая должна быть продублирована в обоих. Здесь нужно обратить внимание на то, что клиент не является безопасной или доверенной средой. По этой причине нам также нужно проверить много вещей на сервере. GraphQL — это пример технологии, которая пытается использовать серверную часть только как «базу данных» без какой-либо бизнес-логики на сервере.

Состояние приложения также больше не находится только на сервере (база данных/сеанс…). Мы также должны поддерживать состояние в клиенте. Мы можем добиться этого с помощью таких решений, как LocalStorage/PouchBD, а также с помощью таких технологий, как Redux, Flux, Reflux и Vuex. Здесь нам также нужно решить, какие данные принадлежат клиенту, какие данные принадлежат серверу, а какие нужно сохранить в обоих местах. Нам также необходимо подумать о (многолидерной) синхронизации: поскольку данные могут быть одновременно изменены более чем на одном клиенте, который может быть отключен в течение более длительного периода времени, обязательно возникнут конфликты записи, которые необходимо разрешить.

Изменение границы между клиентом и сервером также означает, что нам нужно выполнять больше кода JavaScript на клиенте. Время, необходимое браузеру для разбора и выполнения кода JavaScript, многими недооценивается. Это также означает, что страницы с большим количеством JavaScript могут иметь более медленное время запуска. Чтобы избежать этого, многие фреймворки предлагают гидратацию: приложение сначала будет отображаться на сервере с помощью Node и доставляться клиенту в виде HTML. Когда JS загружается, он берет на себя разметку, и с этого момента приложение отображается на клиенте.

Еще одной причиной гидратации является SEO: поскольку JS не может надежно интерпретироваться поисковыми системами, имеет смысл предоставлять контент в виде HTML. Благодаря гидратации мы добавляем не только дополнительную сложность клиенту, но и дополнительную инфраструктуру.

Исполнение в неконтролируемой среде

Браузеры также являются более неконтролируемой средой выполнения, чем наш сервер. Нам нужно обратить на это внимание, когда мы загружаем больше кода в браузер. Один момент заключается в том, что наблюдаемость хуже. На сервере мы можем вести журналы и ловить и регистрировать исключения. Мы также можем сделать это на клиенте, но это намного сложнее и легче ошибиться. Мы также должны обратить внимание на некоторые факторы, касающиеся производительности. Мы часто тестируем наш JavaScript на очень быстрых машинах. Но на мобильных устройствах время, необходимое для обработки JavaScript, очень велико. Подробности можно найти здесь и здесь .

Браузеры также являются более неконтролируемой средой выполнения, чем наш сервер. Нам нужно обратить на это внимание, когда мы загружаем больше кода в браузер.

JavaScript выполняется в одном потоке. Только операции ввода-вывода выполняются асинхронно вне этого потока. Задача ЦП блокирует поток. Таким образом, задачи с интенсивным использованием ЦП приводят к тому, что страница

перестает отвечать на запросы. Если мы выполняем много бизнес-логики, мы тем самым блокируем поток, и страница кажется зависшей. Сегодня мы можем выполнять код в отдельном потоке с помощью Web Workers. Однако это не поддерживается в старых браузерах, что еще больше снижает производительность на старых и медленных устройствах.

Кроме того, неконтролируемая среда выполнения также имеет определенные последствия. На сервере мы можем выбрать версию нашего языка программирования (например, JDK 7). Однако существует невероятное количество веб-браузеров в самых разных версиях. Даже самая новая версия браузера не поддерживает все указанные функции. Поэтому сложность тестирования приложения намного выше.

Таким образом, перенос маршрутизации, шаблонов и логики представления на клиента увеличивает его ответственность. Кроме того, требуется дополнительная инфраструктура, координация, дублирование и косвенность, и мы переносим код из контролируемой среды выполнения в неконтролируемую среду выполнения.

Альтернативная архитектура

Давайте не будем делать шаг назад и сравним архитектуру из предыдущего раздела с классическим приложением с графическим интерфейсом, которое мы будем разрабатывать с помощью .NET, Swing или Eclipse RCP. Мы поймем, что различия довольно малы. Некоторым командам это кажется очень привлекательным, особенно если они еще не разработали никаких веб-приложений. Опыт работы с трехуровневой архитектурой можно перенести в сеть. Средой выполнения больше не является Windows с CLR или JVM, а среда выполнения JavaScript в браузере.

К сожалению, несмотря на свою популярность, этот подход игнорирует реальную силу сети, которая была сформирована декларативными языками и поэтому имеет чрезвычайно устойчивую к ошибкам архитектуру. Это можно проиллюстрировать изменением технологии: браузер представляет собой высокопроизводительный графический движок и является наиболее оптимизированным приложением практически на всех платформах. Он реализован на C, C++ или других низкоуровневых языках, таких как Rust, и максимально использует аппаратное ускорение базовых систем. Мы можем использовать эти возможности, обратившись к ним через API из JavaScript. В качестве альтернативы мы можем использовать декларативные языки HTML и CSS, которые были разработаны специально для этой среды. Низкоуровневый код браузера может парсить,

Поскольку эти языки были созданы именно для этой цели, их производительность оптимальна. Кроме того, они декларативны и особенно

устойчивы к ошибкам в неблагоприятной среде выполнения браузера, описанной выше. Если в коде JavaScript есть ошибка, генерируется исключение, и остальная часть кода не будет выполняться. CSS, с другой стороны, пропускает неизвестные правила, а HTML рассматривает неизвестные теги как элементы. Их декларативный характер также позволяет в значительной степени обеспечить совместимость между версиями браузеров. Вот как страницы HTML и CSS, отображаемые на стороне сервера, могут интерпретироваться старыми и новыми браузерами, и даже устройства, которые не существовали во время создания кода, обычно могут интерпретировать их в той степени, в которой они пригодны для использования.

Те, кто пишет код на стороне сервера, могут рассматривать HTML как конфигурацию компонентов: можно использовать существующие компоненты, такие как текстовые поля и поля ввода, элементы выбора (списки), а также видео и автоматические элементы, без их повторной реализации. Естественно, JavaScript также используется, но он больше не является необходимым компонентом рабочей страницы, а вместо этого является дополнительным элементом, который может улучшить эргономику или внешний вид страницы.

Мы рекомендуем этот подход, основанный на основных функциях Интернета и использовании этих трех ключевых технологий на основе этих основных принципов. По этой причине мы создали веб-сайт с лучшими практиками и дали подходу имя: ROCA (ROCA означает ресурсно-ориентированную клиентскую архитектуру).

Как я могу применить это на практике?

До сих пор мы говорили об абстрактной архитектуре, но как это выглядит на практике? Можем ли мы представить себе веб-приложение без JavaScript? Возможно, мы привыкли получать некоторую структуру JSON с сервера и использовать ее для создания элементов DOM. Но JSON — это не язык Интернета. HTML — это язык Интернета. Почему нам нужно сделать дополнительный шаг, чтобы среда JavaScript могла создавать HTML из нашего JSON? Почти каждый серверный фреймворк включает механизм шаблонов, с помощью которого мы можем напрямую генерировать HTML и отправлять его клиенту. Например, приложение Spring-Boot по умолчанию поставляется в комплекте с Thymeleaf, и мы можем использовать его для прямой доставки HTML-страниц без использования какой-либо инфраструктуры JavaScript.

Заставьте это работать: HTML

Если мы хотим разработать приложение без JavaScript, первым шагом будет создание веб-приложения, состоящего только из HTML. HTML5 включает довольно много элементов с большим количеством функций. Вот три примера:

<input type="date">создает средство выбора даты.

<audio>создает плеер для аудиофайлов

<video>создает проигрыватель для видеофайлов.

Кроме того, для этих элементов всегда есть запасной вариант, чтобы старые браузеры могли продолжать работать. Часто браузер может принимать более правильные решения о том, как отображать эти элементы, чем мы. Приложение <input type="date">на смартфоне оптимизировано таким образом, что пользователи могут выбирать дату пальцем, и по этой причине использовать этот инструмент выбора даты проще, чем инструмент выбора даты из библиотеки JavaScript. Мы также не должны забывать, что HTML предназначен не только для логики представления. С помощью простой ссылки мы можем перейти с одной страницы на другую. HTML-формы также чрезвычайно эффективны и позволяют нам создавать динамические HTTP-запросы на основе пользовательского ввода, которые затем отправляются на сервер. Затем сервер выполняет изменение, запрошенное пользователем (рис. 4).

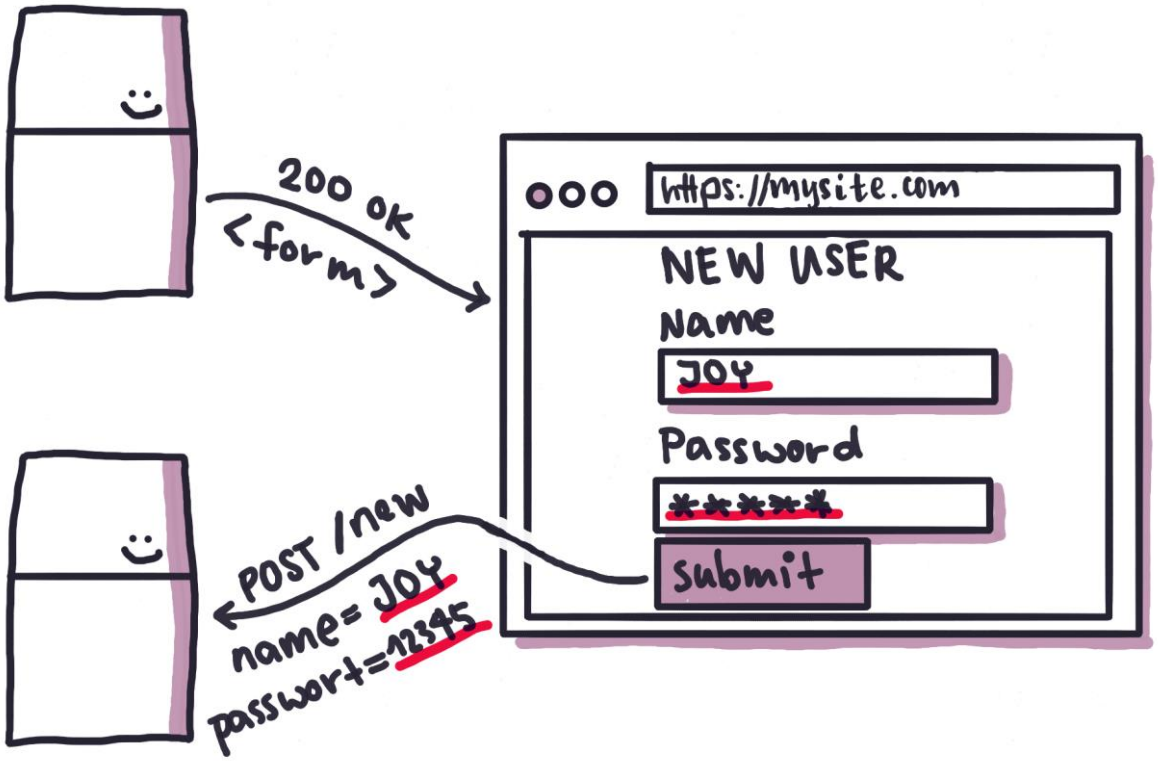

Рисунок 4. HTML-формы позволяют выполнять динамические HTTP-запросы

Иногда мы также хотим показать другую страницу в зависимости от ввода пользователя. Поскольку сервер теперь контролирует функциональность в приложении, он может решить, куда идти дальше с помощью перенаправления (рис. 5).

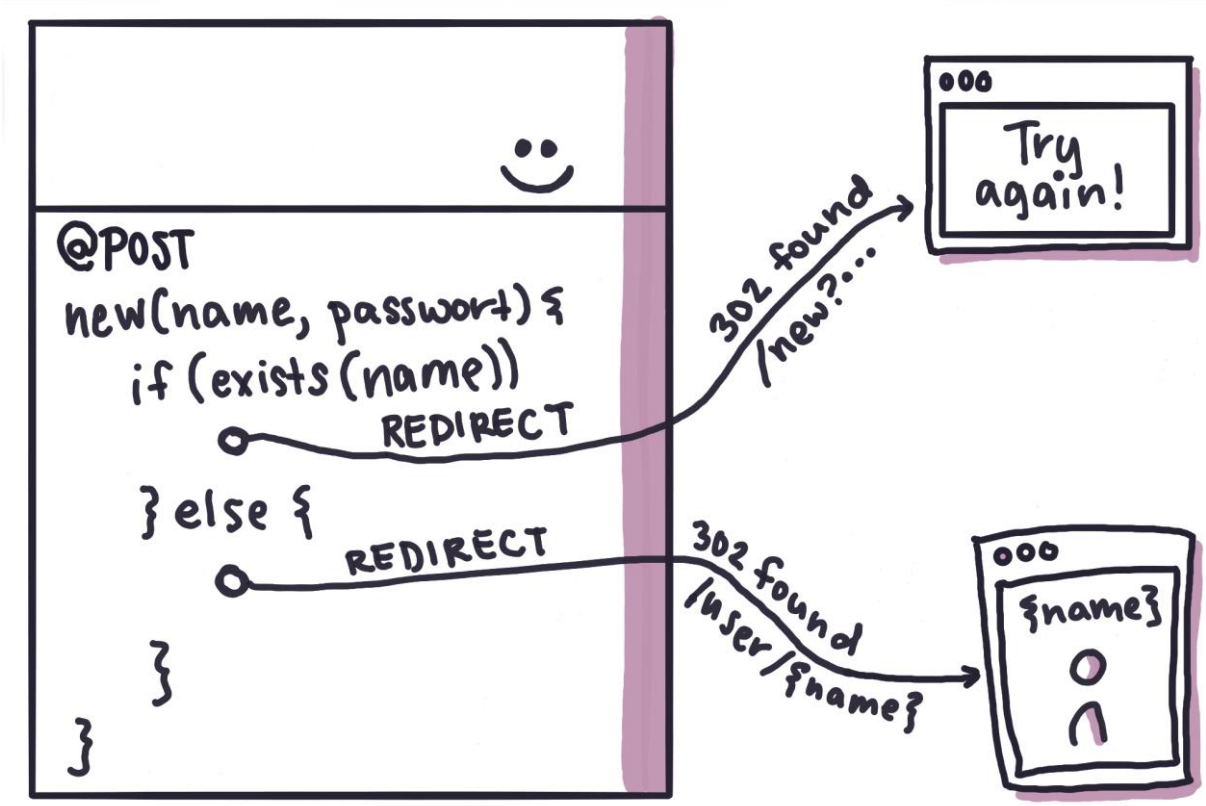

Рисунок 5. Сервер решает, что делать дальше

Одним из основных преимуществ этого подхода является то, что HTML может легко использоваться программами чтения с экрана. Это позволяет относительно легко реализовать специальные возможности в приложении. Проект A11Y — хороший ресурс для дальнейших улучшений. Конечно, также возможно создать доступное приложение, ориентированное на JavaScript, но это требует больших усилий и большого внимания.

Возможность повторного использования

Еще одним преимуществом HTML является то, что мы можем произвольно вкладывать элементы. Чтобы упростить повторное использование элементов пользовательского интерфейса, мы можем извлечь некоторые элементы в компоненты. Компоненты — это многократно используемая структура HTML, которую мы можем использовать в качестве абстракции для нашей функциональности.

Чтобы поддерживать обзор наших компонентов, мы можем структурировать их в системе, подобной Atomic Design. Подробнее о них можно узнать в статье Уте Майер на стр. 58. Здесь более крупные компоненты всегда содержат более мелкие компоненты. Самый простой способ применить это на практике — определить фрагмент HTML, скопировать разметку и отключить содержимое в компоненте. Тем не менее, это не легко поддерживать с течением

времени. Большинство движков шаблонов предоставляют способ определить фрагмент HTML один раз и сделать его многоразовым. В Thymeleaf мы можем сделать это с помощью th:replaceи th:insert.

Одним из основных преимуществ этого подхода является также то, что мы можем стилизовать наши компоненты на основе этой HTML-структуры. Здесь начинаются более серьезные проблемы.

Сделать красиво: CSS

Теперь мы подошли к моменту, когда мы можем потерпеть неудачу. С одной стороны, можно утверждать, что важнее всего, чтобы приложение работало, даже если оно выглядит не очень красиво. С другой стороны, никто не хочет использовать уродливое приложение. В этом контексте важна определенная эстетика.

Эстетика для Интернета

Хотя «красота» субъективна, мы можем потратить много времени на такие темы, как дизайн или типографика, чтобы развить чувство того, как все должно выглядеть в готовом продукте. К сожалению, у многих разработчиков нет необходимого времени или желания по-настоящему вникать в тему.

К счастью, нам не всегда приходится изобретать велосипед. Можно использовать готовые библиотеки компонентов вроде Bootstrap или Foundation . Здесь дизайнеры уже приняли самые важные решения, чтобы приложение было удобно использовать конечным пользователям.

Для тех, кто хочет изменить библиотеку компонентов и расширить ее дополнительными компонентами, очень полезен набор инструментов библиотеки шаблонов, такой как Fractal . Это особенно полезно, когда компоненты должны повторно использоваться между проектами. Для больших проектов CSS мы рекомендуем использовать препроцессор, такой как Sass, чтобы CSS можно было разделить на несколько файлов. Затем CSS для определенного компонента HTML можно разместить непосредственно рядом с компонентом в той же папке. Здесь мы можем использовать соглашение об именах, такое как БЭМ для компонентов CSS. Конвейер активов, такой как faucet-pipeline , может использоваться для компиляции CSS. Когда мы также пишем JavaScript для нашего приложения, мы также можем использовать кран для этого.

Сделайте это быстро: JavaScript

Мы достигли точки, на которой можно остановиться: у нас есть работающее веб-приложение, которое выглядит красиво. Однако теперь мы можем оптимизировать и улучшить наше приложение с помощью JavaScript. Причина, по которой SPA часто кажется быстрее, заключается в том, что браузеру не нужно

анализировать и оценивать CSS и JavaScript при каждом переходе страницы. С такой библиотекой, как Turbolinks , мы можем добиться именно этого: когда пользователь нажимает на ссылку, HTML на странице заменяется без перезагрузки CSS и JavaScript.

С HTML-формами мы можем добиться чего-то подобного: вместо того, чтобы отправлять браузеру содержимое формы и извлекать результат в виде новой страницы, мы можем отправить форму через Ajax и работать с возвращаемым HTML-кодом, заменяя только части страницы. .

Нет необходимости создавать одностраничное приложение, чтобы избежать повторного анализа CSS и JavaScript при каждой загрузке страницы.

Чтобы объединить содержимое нескольких страниц в браузере, мы можем трансклюдировать ссылки — это означает, что ссылка заменяется ее содержимым. Для этого некоторый общий код JavaScript может «следовать» по ссылке — например, использовать Ajax для отправки запроса на сервер — и включать результат непосредственно на страницу.

Это особенно привлекательно, потому что во всех этих трех случаях эффект аналогичен эффекту одностраничного приложения, а переходы страниц работают плавно. Однако страница работает и тогда, когда JavaScript отсутствует, глючит, еще не загружен или по каким-то другим причинам не может быть выполнен.

Для улучшения JavaScript в нашем приложении теперь мы можем использовать пользовательские элементы для определения наших собственных элементов HTML в браузере:

```
class MyComponent extends HTMLElement {
 connectedCallback() {
 // instantiate component
 }
}
customElements.define("my-component", MyComponent)
```

Преимущество здесь в том, что логика, определенная в connectedCallbackфункции, всегда будет выполняться браузером при создании <my-component>элемента в HTML DOM. Это также работает вместе с Turbolinks или включением, потому что браузер заботится о создании экземпляра JavaScript, как только элемент появляется в DOM.

СПА против РПЦ

На практике мы убедились, что когда мы разрабатываем приложения с серверным рендерингом HTML и пользовательскими элементами, тщательно модульным CSS и небольшим количеством JavaScript, мы можем создавать архитектурно чистые, компактные, быстрые и эргономичные приложения. Мы не боимся сравнения этих приложений с приложениями, созданными с использованием популярных подходов SPA — во многих аспектах они часто превосходят их. Мы обнаружили, что они не только имеют меньшие коммуникационные издержки, более высокую отказоустойчивость и более легко доступны, но и их легче поддерживать в долгосрочной перспективе.Мы зависим от браузера и его стандартов, а не от фреймворка SPA, который может полностью измениться в следующей версии или может быть заменен следующей альтернативной фреймворком.

СПИСОК ЛИТЕРАТУРЫ

1. Zachary Kessin. Programming HTML5 Applications, Building Powerful Cross-Platform Environments in Ja- vaScript. Sebastopol CA, 2011.
2. Andrew Lunny. PhoneGap Beginner's Guide. Birmingham, 2011.
3. The Node Ahead: JavaScript leaps from browser into future. The Register. [Электронный ресурс]. – Режим доступа: http://www.theregister.co.uk/2011/03/01/the_rise_and_rise_of_node_dot_js/. – Дата доступа: 05.09.2012.

4. John Paul Mueller. Professional IronPython, Wrox Professional Guides. Chichester, 2010
5. Boydlee Pollentine. Appcelerator Titanium Smartphone App Development Cookbook. Birmingham, 2011.