# ВВЕДЕНИЕ

Уже несколько лет стандартом и хорошим тоном веб-программирования являются Single Page Application (далее SPA) то есть одностраничные приложения. Это объясняется их очевидными преимуществами:

Запрашивание у сервера только "голых" данных вместо целой страницы.

Возможность непрерывного исполнения Java Script кода на протяжении всего сеанса (например, прослушивание музыки в режиме онлайн).

Более отчетливое разделение на backend и frontend, что помогает легче, быстрее и качественнее проводить отладку приложения. Один и тот же backend может использоваться веб, мобильными и десктоп приложениями, а это в свою очередь экономит время отладки.

В задачах где не требуется загрузка данных приложение легко может работать офлайн.

Без SPA не существовало бы настолько быстрого обмена сообщениями, например, на таких ресурсах как facebook.com, ok.ru, vk.com и т.д. Но данный метод имеет ряд серьезных технических упущений. Веб приложение всегда будет загружаться целиком. Даже если некий функционал по предусмотренной внутренней логике никогда не будет востребован пользователем так или иначе обязательно загрузится вместе со всем приложением. В данной курсовой работе представлен способ избежать загрузки этого нежелательного балласта с помощью Lazy-loading. Lazy-Loading (в переводе с англ. ленивая загрузка) - способ отложенной загрузки частей приложения. Если, кратко то в основе этого способа лежит деление приложения на некие модули, которые и будут подгружаться по необходимости. Благодаря этому способу можно больше не беспокоиться о загрузке лишнего контента, который замедляет её и наконец-то навсегда забыть о старых методах веб программирования. Это технологии будущего, которые уже давно являются чем-то обыденным для таких гигантов как Google, MicroSoft и тд. Кстати, о Google и MicroSoft. Для этого будет использоваться фреймворк Angular (не путать с Angular.js), который является коллаборацией двух мировых гигантов.

Angular написан на языке TypeScript (являющийся продуктом MicroSoft) разработчиками Google. Одно из главных преимуществ TypeScript это его компиляция в Java Script код читаемый даже в старых версиях браузеров. TypeScript отличается от JavaScript возможностью явного статического назначения типов, поддержкой использования полноценных классов (как в традиционных объектно-ориентированных языках), а также поддержкой подключения модулей, что призвано повысить скорость разработки. Angular это хорошо оптимизированный фреймворк, гибкий, с возможностью подключения любых дополнительных сервисов. Он построен на модели MVC (Model View Control). Эта схема разработки приложения очень удобна, тем что разделяет данных приложения, пользовательского интерфейса и управляющей логики на три отдельных компонента: модель, представление и контроллер - таким образом, что модификация каждого компонента может осуществляться независимо.

Однако фреймворк Angular собирает приложение в сплошной JavaScript код, что является большой проблемой для поисковых роботов SEO. Собранный для публикации проект Angular хоть и содержит множество технологических решений, тем не менее при загрузке в браузер представляет собой практически пустую HTML страницу, за исключением разных JavaScript и CSS подключений, и главного компонента с которого Angular начинает взаимодействие с пользователем. И хотя поисковые роботы уже научились парсить JavaScript код, всё же такой подход не рекомендуется к использованию в веб-приложениях, ориентированных на быстрый поисковый отклик.

Для создания SEO ориентированного продукта, в Angular существует технология генерации HTML-шаблонов на сервере - Server-Side Rendering (SSR). В контексте Angular разработки данная технология носит название - Angular Universal. Angular Universal способствует более эффективной работе поисковых роботов, поскольку они работают с уже сгенерированной страницей. Так же хорошим бонусом является более быстрая работа приложения на мобильных и слабых по производительности устройствах.

Большие IT-компании часто занимаются брендированием своих продуктов. Под этим подразумевается: одинаковые кнопки, элементы страницы, логотипы, виджеты, чаты и т.п. Но при этом из-за большого количества команд и разработчиков внутри компании, и как следствие написание продуктов на разных языках, не представляется возможным использование универсальных библиотек с этими компонентами. Разве, что написание библиотек на каждом из разрабатываемых языков. Но это решение обладает рядом существенных недостатков:

* Не гарантирует полностью идентичных решений;
* Занимает в разы больше времени на разработку;
* Доставляет большие сложности при обновлении библиотек и как следствие проектов в которые они внедрены.

Кажется, было бы хорошо иметь в своем пользовании некую универсальную библиотеку для подобных решений. Библиотеку, которая обладала бы следующими свойствами:

* Легко внедряться в любой веб-проект;
* Простая в использовании внутри проекта;
* Обновляться независимо от разрабатываемого проекта;
* Быть изолированной от случайного вмешательства в стили и логику компонентов внутри неё;

Для создание таких универсальных решений были придуманы веб-компонеты. Веб-компоненты поддерживаются веб-браузерами напрямую и не требуют дополнительных библиотек для работы. Это технология, которая позволяет создавать многократно используемые компоненты в веб-документах и веб-приложениях. Модель веб-компонентов подразумевает инкапсуляцию и совместимость отдельных HTML-элементов. На данный момент частичная поддержка существует в браузерах Chrome, Firefox, Opera и Safari. Для браузеров, не поддерживающих веб-компоненты реализованы полифилы. Веб-компоненты включают три технологии, каждая из которых может использоваться отдельно от других: Custom Elements, HTML Templates, Shadow DOM.

Веб-компоненты имеют свою реализацию в Angular и носит название Angular Elements. Данная технология позволяет использовать компоненты Angular не только в приложениях Angular, но и в любых других веб-приложениях. Это компоненты, которые умеют: делать свои подсчеты, отправлять запросы, принимать значения от приложения через атрибуты, возвращать события и т.д. Иными словами это часть кода из другого приложения, которая работает везде. И всё, что необходимо для это сделать – объявить ранее созданный пользовательский тэг в разрабатываемом проекте.

Цель работы является исследование фреймворка Angular с целью внедрения технологий Lazy-Loading, SSR, Web-components и разработка демонстрационного приложения. Область исследования включает в себя технологии используемые в процессе разработки frontend на языке TypeScript необходимые для Lazy-Loading.

Достижение целей предполагает:

* Анализ специальных информационных источников;
* Исследование фреймворка Angular;
* Исследование Lazy-Loading и сопутствующие технологии;
* Исследование Angular Universal и сопутствующие технологии;
* Исследование Angular Elements и сопутствующие технологии;
* Исследование необходимых составляющих полноценного приложения на Angular: Material Design и тд;
* Разработка демонстрационного приложение.

Работа состоит из: введение, заключение, 3-х глав и библиографии.

В первой главе представлено общее понимание технологии SSR и основы Angular Universal.

Во второй главе описывается теоретические основы фреймворка Angular и технологии Lazy-Loading.

В третьей главе описывается общее понимание Веб-компонентов и Angular Elements

Работа состоит из \_ страниц, \_ рисунков, \_ источников

# Рендеринг на стороне сервера (SSR)

## Виды рендеринга сайтов

Разработчики, часто сталкиваются с решениями, которые влияют на всю архитектуру приложений. Одно из основных решений, которые должны принять веб-разработчики, - это где реализовать логику и рендеринг в своих приложениях. Это может быть сложно, так как существует несколько разных способов создания сайта. Чтобы лучше понять архитектуры, которые выбираются, при принятии этих решение, необходимо иметь четкое понимание каждого подхода и единую терминологию, которую следует использовать при разговоре о них. Различия между этими подходами помогают проиллюстрировать компромиссы рендеринга в сети через призму производительности.

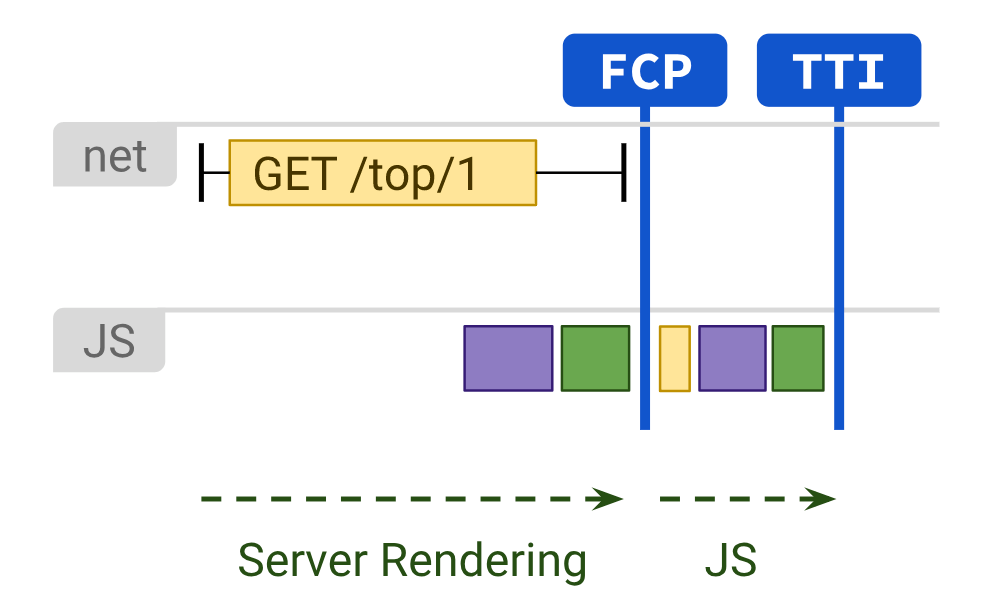
**Терминология рендеринга:**

* **SSR -** рендеринг на стороне сервера - рендеринг клиентского или универсального приложения в HTML на сервере.
* **CSR -** рендеринг на стороне клиента - рендеринг приложения в браузере, обычно с использованием DOM.
* **Регидратация -** «загрузка» представлений JavaScript на клиенте так, чтобы они повторно использовали дерево DOM и данные HTML, представленные сервером.
* **Предварительный рендеринг -** запуск приложения на стороне клиента во время сборки для захвата его исходного состояния в виде статического HTML.

**Терминология производительности:**

* **TTFB (Time To First Byte)** - рассматривается как время между нажатием на ссылку и первым поступающим контентом.
* **FP (First Paint)** - первый раз, когда любой пиксель становится видимым для пользователя.
* **FCP** **(First Contentful Paint)** - время, когда запрашиваемый контент (тело статьи и т. Д.) Становится видимым.
* **TTI (Time To Interactive)** - время, когда страница становится интерактивной (события связаны и т. Д.).

**Серверный рендеринг -** генерирует полный HTML для страницы на сервере в ответ на навигацию. Это позволяет избежать дополнительных циклов обработки данных и шаблонов на клиенте, поскольку они обрабатываются до того, как браузер получает ответ. Серверный рендеринг обычно производит быструю First Paint и First Contentful Paint. Выполнение логики страницы и рендеринга на сервере позволяет избежать отправки большого количества JavaScript клиенту, что помогает быстро достичь Time to Interactive. Это имеет смысл, поскольку при серверном рендеринге просто отправляется текст и ссылки в браузер пользователя. Этот подход может хорошо работать для широкого спектра устройств и условий сети, и открывает интересные оптимизации браузера, такие как потоковый анализ документов.



При использовании серверного рендеринга пользователи вряд ли будут ждать, пока обработается привязанный к процессору JavaScript, прежде чем они смогут использовать сайт. Даже когда нельзя избежать [стороннего JS](https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/loading-third-party-javascript/), использование серверного рендеринга для сокращения собственных [затрат на JS](https://medium.com/@addyosmani/the-cost-of-javascript-in-2018-7d8950fbb5d4) может дать вам больше «[бюджета](https://medium.com/@addyosmani/start-performance-budgeting-dabde04cf6a3)» на все остальное. Однако у этого подхода есть один главный недостаток: генерация страниц на сервере требует времени, что часто может привести к более долгому [времени до первого байта](https://developers.google.com/web/updates/2019/02/TTFB). Достаточно ли серверного рендеринга для вашего приложения, во многом зависит от него самого. Существует давняя дискуссия о правильном применении серверного рендеринга по сравнению с рендерингом на стороне клиента, но важно помнить, что можне использовать серверный рендеринг для одних страниц, а не для других нет. Некоторые сайты успешно применяют гибридные методы рендеринга. Сервер [Netflix](https://medium.com/dev-channel/a-netflix-web-performance-case-study-c0bcde26a9d9) отображает свои относительно статичные целевые страницы, [предварительно выбирая](https://dev.to/addyosmani/speed-up-next-page-navigations-with-prefetching-4285) JS для страниц с интенсивным взаимодействием, предоставляя этим более тяжелым страницам, отображаемым клиентом, более высокую вероятность быстрой загрузки. Многие современные фреймворки, библиотеки и архитектуры позволяют отображать одно и то же приложение как на клиенте, так и на сервере. Эти методы могут использоваться для серверного рендеринга, однако важно отметить, что архитектуры, в которых рендеринг происходит как на сервере, так и на клиенте, представляют собой собственный класс решений с очень разными характеристиками производительности и компромиссами. Пользователи React могут использовать [renderToString()](https://reactjs.org/docs/react-dom-server.html) или решения, построенные на его основе, такие как [Next.js,](https://nextjs.org/) для серверного рендеринга. Пользователи Vue могут взглянуть на [руководство по серверному рендерингу](https://ssr.vuejs.org/) Vue или [Nuxt](https://nuxtjs.org/). Для Angular есть Angular [Universal](https://angular.io/guide/universal). В большинстве популярных решений используется некоторая форма гидратации, поэтому перед выбором инструмента ознакомьтесь с подходом, который используется.

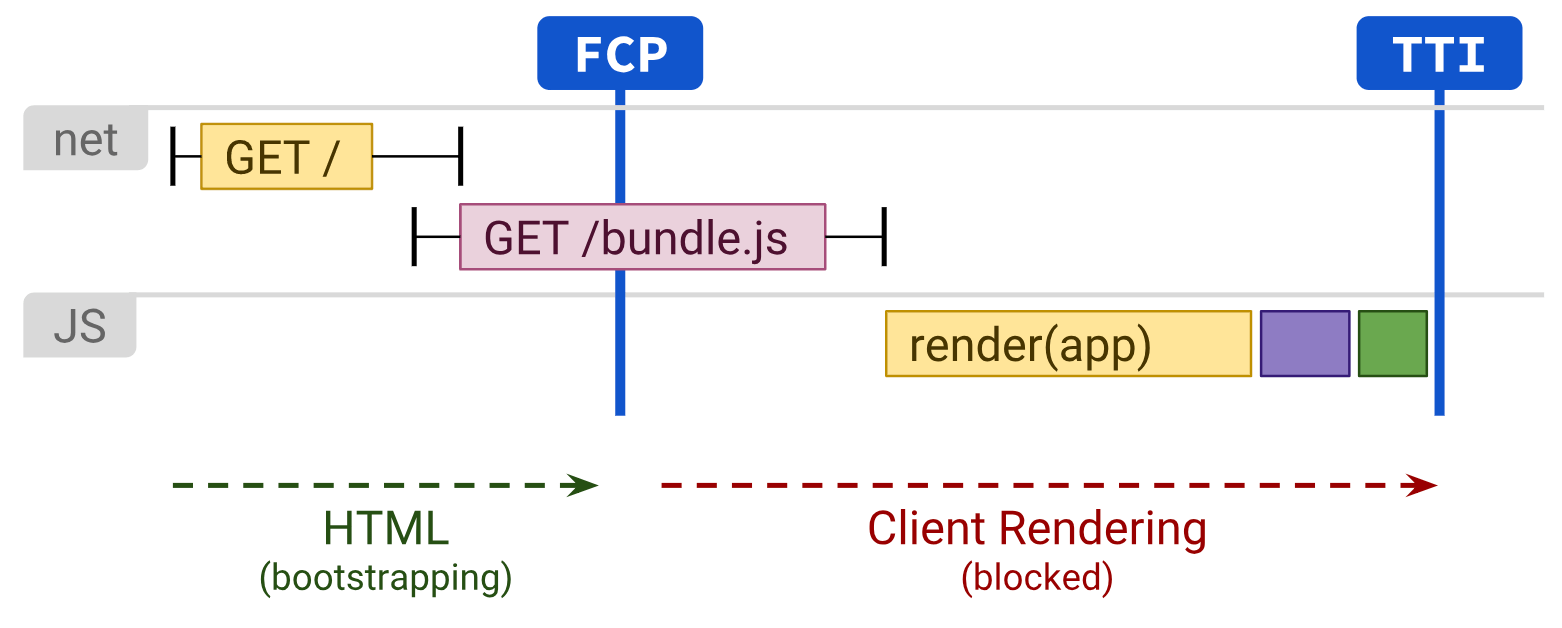
**Статический рендеринг -** происходит во время сборки и предлагает быстрые First Paint, First Contentful Paint и Time To Interactive - при условии, что количество JS на стороне клиента ограничено. В отличие от серверного рендеринга, ему также удается достичь стабильно быстрого времени до первого байта, поскольку HTML-код для страницы не нужно генерировать на лету. Как правило, статический рендеринг означает создание отдельного HTML-файла для каждого URL-адреса заранее. Поскольку HTML-ответы генерируются заранее, статические рендеры могут быть развернуты на нескольких CDN, чтобы воспользоваться преимуществом пограничного кэширования (edge-caching).

Diagram showing static rendering and optional JS execution affecting FCP
and TTI

Решения для статического рендеринга бывают всех форм и размеров. Такие инструменты как [Gatsby](https://www.gatsbyjs.org/) спроектированы так, чтобы разработчики почувствовали, что их приложения рендерятся динамически быстрее чем генерируются на этапе сборки. Другие, как [Jekyll](https://jekyllrb.com/) и [Metalsmith](https://metalsmith.io/) принимают их статическую природу, обеспечивая более шаблонный подход. Одним из недостатков статического рендеринга является то, что отдельные HTML-файлы должны создаваться для каждого возможного URL. Это может быть сложно или даже невозможно, если не представляется возможным предсказать, какие будут эти URL-адреса раньше времени, или для сайтов с большим количеством уникальных страниц. Пользователи React могут быть знакомы с [Gatsby](https://www.gatsbyjs.org/), [статическим экспортом Next.js](https://nextjs.org/learn/excel/static-html-export/) или [Navi](https://frontarm.com/navi/) - все это делает его удобным для автора с использованием компонентов. Тем не менее, важно понимать разницу между статическим рендерингом и предварительным рендерингом: статический рендеринг страниц являются интерактивными без необходимости выполнения большого количества JS на стороне клиента, тогда как предварительный рендеринг улучшает First Paint или First Contentful Paint одностраничного приложения (SPA), которое необходимо загрузить клиент для того, чтобы страницы были по-настоящему интерактивными. Если нет уверенности, является ли данное решение статическим или предварительным рендерингом, можно попробовать следующий тест: отключите JavaScript и загрузите созданные веб-страницы. Для статически визуализированных страниц большая часть функциональности будет по-прежнему существовать без включенного JavaScript. Для предварительно обработанных страниц могут существовать некоторые базовые функции, такие как ссылки, но большая часть страницы будет инертной. Еще один полезный тест: замедление работы сети с помощью Chrome DevTools и наблюдение за загрузкой JavaScript до того, как страница станет интерактивной. Для предварительного рендеринга обычно требуется больше JavaScript, чтобы стать интерактивным, и этот JavaScript имеет тенденцию быть более сложным, чем подход [прогрессивного улучшения,](https://developer.mozilla.org/en-US/docs/Glossary/Progressive_Enhancement) используемый при статическом рендеринге.

Серверный Рендеринг не является самым оптимальным решением. Его динамическая природа может сопровождаться [значительными вычислительными накладными](https://medium.com/airbnb-engineering/operationalizing-node-js-for-server-side-rendering-c5ba718acfc9) расходами. Многие решения для рендеринга серверов не сбрасываются рано, могут задержать TTFB или удвоить отправку данных (например, встроенное состояние, используемое JS на клиенте). В React, функция renderToString() может быть медленной, поскольку она является синхронной и однопоточной. Получение серверным рендерингом «прав» может включать в себя нахождение или создание решения для [компонентов кэширования](https://medium.com/@reactcomponentcaching/speedier-server-side-rendering-in-react-16-with-component-caching-e8aa677929b1) , управление потреблением памяти, применение техник [мемоизации](https://speakerdeck.com/maxnajim/hastening-react-ssr-with-component-memoization-and-templatization), и многие другие проблемы. Обычно обрабатывается/перестраивается одно и то же приложение несколько раз - один раз на клиенте и один раз на сервере. Тот факт, что при рендеринге с сервера может появиться что-то раньше, не означает, что на клиенте меньше работы. Серверный рендеринг генерирует HTML по требованию для каждого URL, но может быть медленнее, чем просто обслуживание статического рендеринга контента. Если можно добавить дополнительную работу, серверный рендеринг в совокупности с [кэшированием HTML](https://freecontent.manning.com/caching-in-react/) может значительно сократить время серверного рендеринга. Преимуществом рендеринга на сервере является возможность извлекать больше «живых» данных и отвечать на более полный набор запросов, чем это возможно при статическом рендеринге. Страницы, требующие персонализации, являются конкретным примером типа запроса, который не будет хорошо работать при статическом рендеринге. Серверный рендеринг также может представлять интересные решения при построении [PWA](https://developers.google.com/web/progressive-web-apps/). Лучше использовать полностраничное [сервис-воркер (service-worker)](https://developers.google.com/web/fundamentals/primers/service-workers/) кэширование или просто рендерить отдельные части контента на сервере?

**Рендеринг на стороне клиента (CSR)** - означает рендеринг страниц непосредственно в браузере с использованием JavaScript. Вся логика, выборка данных, шаблоны и маршрутизация обрабатываются на клиенте, а не на сервере. Рендеринг на стороне клиента может быть трудно получить и быстро сохранить для мобильных устройств. Он может приблизиться к производительности серверного рендеринга, если выполняет минимальную работу, сохраняя [жесткий бюджет JavaScript](https://mobile.twitter.com/HenrikJoreteg/status/1039744716210950144) и предоставляя ценность в минимально возможном количестве [RTTs](https://en.wikipedia.org/wiki/Round-trip_delay_time). Критические сценарии и данные могут быть доставлены быстрее, используя [HTTP/2 Server Push](https://www.smashingmagazine.com/2017/04/guide-http2-server-push/) или <link rel=preload>, что заставляет парсер работать на вас быстрее. Шаблоны, такие как [PRPL](https://developers.google.com/web/fundamentals/performance/prpl-pattern/), стоит оценить, чтобы обеспечить мгновенную начальную и последующую навигацию.



Основным недостатком рендеринга на стороне клиента является то, что количество требуемого JavaScript имеет тенденцию к росту по мере роста приложения. Это становится особенно трудным с добавлением новых библиотек JavaScript, полифилов и стороннего кода, которые конкурируют за вычислительную мощность и часто должны обрабатываться до того, как содержимое страницы может быть отображено. Опыт работы с CSR, основанный на больших пакетах JavaScript, должен учитывать [агрессивное разделение кода](https://developers.google.com/web/fundamentals/performance/optimizing-javascript/code-splitting/) и обязательно загружать JavaScript - «обслуживайте только то, что вам нужно, когда вам это нужно». Для случаев, когда интерактивность незначительна или отсутствует, рендеринг сервера может представлять собой более масштабируемое решение этих проблем. Для разработчиков, создающих одностраничное приложение, идентификация основных частей пользовательского интерфейса, используемых большинством страниц, означает, что можно применить технику [кэширования Application Shell](https://developers.google.com/web/updates/2015/11/app-shell) . В сочетании с сервис-воркерами (Service Workers) это может значительно улучшить воспринимаемую производительность при повторных посещениях.

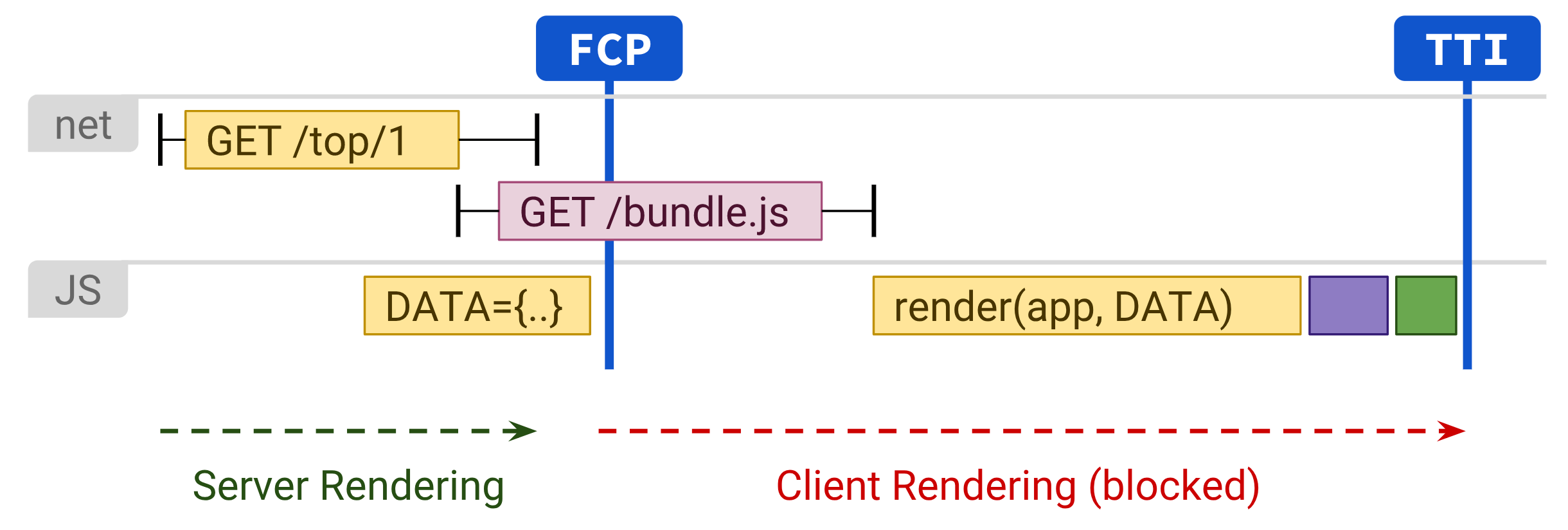
**Объединение серверного рендеринга и CSR через регидратацию -** часто называемый универсальным рендерингом или просто «SSR», этот подход пытается сгладить углы между рендерингом на стороне клиента и рендерингом сервера, выполняя оба действия. Запросы навигации, такие как полная загрузка или перезагрузка страницы, обрабатываются сервером, который отображает приложение в HTML, затем JavaScript и данные, используемые для визуализации, встраиваются в итоговый документ. При аккуратной реализации это обеспечивает быструю First Contentful Paint точно так же, как серверный рендеринг, а затем «подхватывает», снова выполняя рендеринг на клиенте, используя технику, называемую [(ре)гидратация](https://docs.electrode.io/guides/general/server-side-data-hydration). Это новое решение, но оно может иметь некоторые существенные недостатки производительности.

Основным недостатком SSR с регидратацией является то, что он может оказать существенное негативное влияние на Time To Interactive, даже если он улучшит First Paint. Страницы SSR часто выглядят обманчиво загруженными и интерактивными, но на самом деле не могут реагировать на ввод, пока JS на стороне клиента не будет выполнен и обработчики событий не присоединены. Это может занять несколько секунд или даже минут на мобильном телефоне.

HTML document
containing serialized UI, inlined data and a bundle.js script

Проблемы регидратации часто могут быть хуже, чем замедленная интерактивность из-за JS. Чтобы клиентский JavaScript мог точно «подхватить», где сервер остановился, без необходимости повторного запроса всех данных, которые сервер использовал для визуализации своего HTML, текущие SSR решения обычно сериализуют ответ от пользовательского интерфейса. зависимости данных в документе в виде тегов скрипта. Полученный HTML-документ содержит высокий уровень дублирования (рис. ^). Как можно наблюдать, сервер возвращает описание пользовательского интерфейса приложения в ответ на запрос навигации, но он также возвращает исходные данные, использованные для создания этого пользовательского интерфейса, и полную копию реализации пользовательского интерфейса, которая затем загружается на клиенте. Только после завершения загрузки и выполнения bundle.js этот интерфейс становится интерактивным.

Метрики производительности, собранные с реальных веб-сайтов с использованием регидратации SSR, указывают на то, что его использование не рекомендуется. В конечном счете, причина кроется в пользовательском опыте: в конечном итоге крайне просто оставить пользователей в «странной долине».



Однако есть надежда на SSR с регидратацией. В краткосрочной перспективе только использование SSR для контента с высокой степенью кэширования может уменьшить задержку TTFB, что дает результаты, аналогичные предварительному рендерингу. Регидратация [постепенно](https://www.emberjs.com/blog/2017/10/10/glimmer-progress-report.html) или частично может стать ключом к повышению эффективности этого метода в будущем.

**Рендеринг потокового сервера и прогрессивная регидратация –** ещё один способ рендеринга. [Рендеринг потокового сервера](https://zeit.co/blog/streaming-server-rendering-at-spectrum) позволяет отправлять HTML порциями, которые браузер может визуализировать по мере получения. Это может обеспечить быструю First Paint и First Contentful Paint, поскольку разметка поступает к пользователям быстрее. В React потоки, являющиеся асинхронными в [renderToNodeStream()](https://reactjs.org/docs/react-dom-server.html" \l "rendertonodestream) - по сравнению с синхронным renderToString() - означают, что обратное давление хорошо обрабатывается. Прогрессивная регидратация также стоит того, чтобы за ней следить, и кое-что, что [изучал](https://github.com/facebook/react/pull/14717) React. При таком подходе отдельные части приложения, отображаемого на сервере, «загружаются» с течением времени, а не по общему текущему подходу - инициализации всего приложения сразу. Это может помочь уменьшить объем JavaScript, необходимый для того, чтобы сделать страницы интерактивными, поскольку обновление на стороне клиента низкоприоритетных частей страницы может быть отложено для предотвращения блокировки основного потока. Это также может помочь избежать одной из самых распространенных ошибок регидратации в SSR, когда дерево DOM, отображаемое сервером, разрушается, а затем немедленно перестраивается - чаще всего потому, что при первоначальной синхронной визуализации на стороне клиента требуются не совсем готовые данные, возможно, ожидающие разрешения Promise.

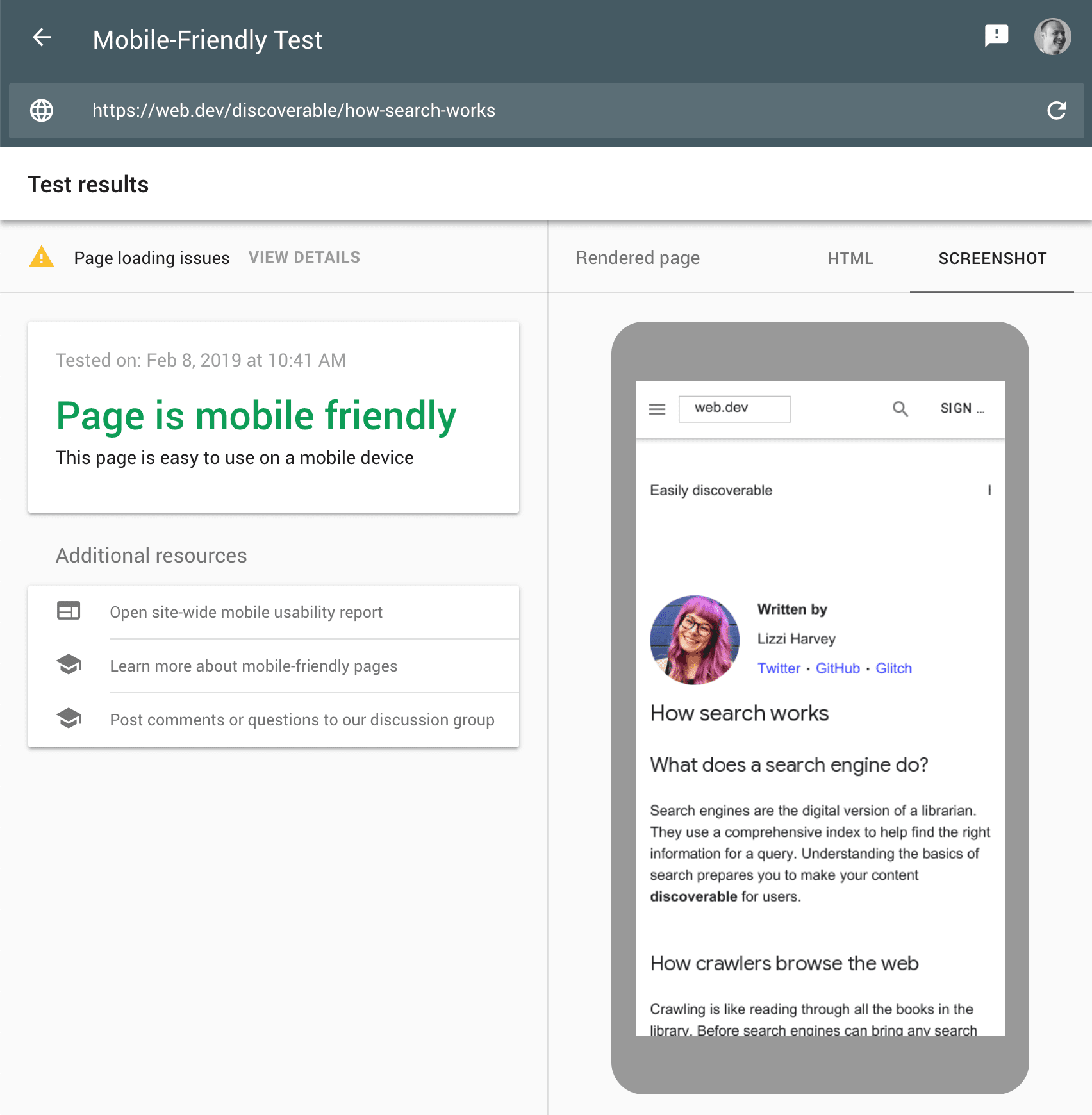
**Частичная регидратация -** оказалась трудно осуществимой. Этот подход является продолжением идеи прогрессивной регидратации, где анализируются отдельные части (компоненты/виды/деревья), которые должны быть постепенно регидратированы, и идентифицируются те, у которых мало интерактивности или нет реактивности. Для каждой из этих в основном статических частей соответствующий код JavaScript затем преобразуется в инертные ссылки и декоративную функциональность, сокращая их площадь на стороне клиента почти до нуля. Подход частичной гидратации имеет свои проблемы и недостатки. Это создает некоторые интересные проблемы для кэширования, и навигация на стороне клиента означает, что мы не можем предполагать, что серверный HTML-код для инертных частей приложения будет доступен без полной загрузки страницы.

**Трисоморфный рендеринг -** также может представлять интерес в случае если [сервис-воркеры](https://developers.google.com/web/fundamentals/primers/service-workers/) (Service Workers) являются подходящим вариантом для приложения. Это метод, при котором возможно использовать потоковый рендеринг сервера для начальной/не-JS-навигации, а затем «попросить» сервис-воркер собрать HTML для навигации после его установки.

Diagram of Trisomorphic rendering, showing a browser and service worker
communicating with the server

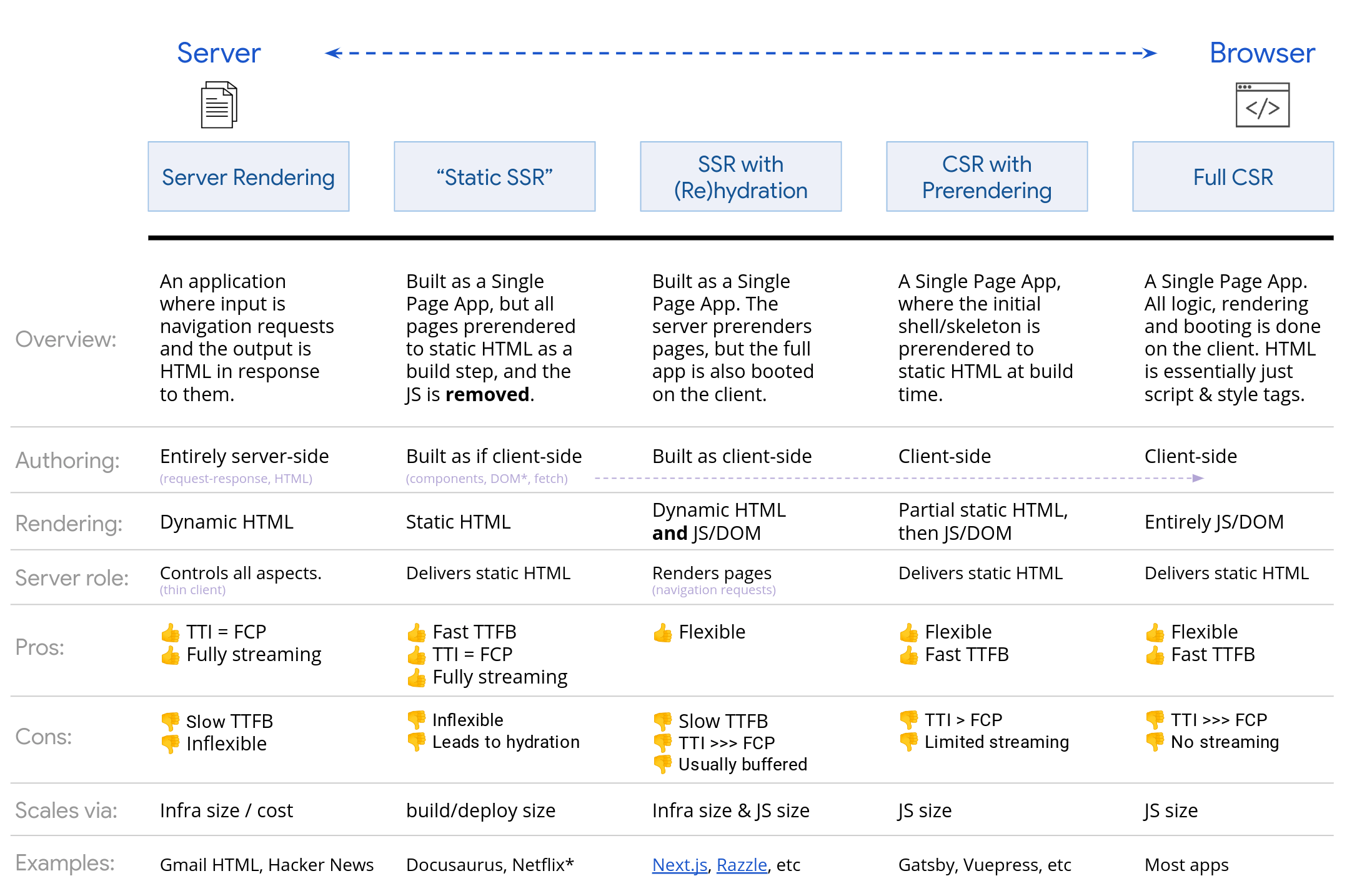
Это может поддерживать кэшированные компоненты и шаблоны в актуальном состоянии и обеспечивает навигацию в стиле SPA для отображения новых представлений в одном сеансе. Этот подход работает лучше всего, когда возможно совместно использовать один и тот же код шаблонов и маршрутизации между сервером, клиентской страницей и сервис-воркером.

Команды часто учитывают влияние SEO при выборе стратегии рендеринга в сети. Рендеринг сервера часто выбирается для обеспечения «полного вида», который сканеры могут легко интерпретировать. Сканеры [могут понимать JavaScript](https://web.dev/discoverable/how-search-works), но часто есть [ограничения, о которых](https://developers.google.com/search/docs/guides/rendering) стоит знать, как они рендерятся. Рендеринг на стороне клиента может работать, но часто не без дополнительного тестирования и работы. В последнее время [динамический рендеринг](https://developers.google.com/search/docs/guides/dynamic-rendering) также стал вариантом, заслуживающим внимания, если ваша архитектура сильно зависит от клиентского JavaScript.



В случае сомнений инструмент [Mobile Friendly Test](https://search.google.com/test/mobile-friendly) неоценим для проверки того, что выбранный подход делает то, на что надеется разработчик (рис. ^). Он показывает визуальный предварительный просмотр того, как какая-либо страница отображается сканеру Google, найденный сериализованный контент HTML (после выполнения JavaScript) и любые ошибки, обнаруженные во время рендеринга.

При выборе подхода к визуализации следует измерить и понять, какие существуют слабые места.



Необходимо подумать над тем, может ли статический рендеринг или рендеринг сервера удовлетворить 90% ожидаемого результата. Вот удобная инфографика, показывающая спектр сервер-клиент (рис. ^). Совершенно нормально в основном отправлять HTML с минимальным JS, чтобы получить интерактивный опыт.

## Angular и SEO

Когда дело доходит до SEO в проектах Angular приходится сталкиваться с дополнительными проблемами. К счастью, [Angular Universal](https://searchengines.guru/go?link=https://angular.io/guide/universal" \t "_blank" \o "https://angular.io/guide/universal) позволяет достаточно легко их решить. При этом стоит иметь в виду, что «легко» – это относительный термин в данном случае. Работа с Angular Universal требует определённых технических знаний и навыков. Почти наверняка придётся привлекать к этому процессу команду разработчиков. Но когда все манипуляции закончатся, сайт будет предоставлять оптимизированные страницы, которые поисковые системы смогут легко находить и индексировать. В данной главе разъясняется, что представляет собой Angular Universal и почему о нём важно знать оптимизаторам.

Angular – это фантастический фрейморк для создания модульных, удобных для пользователей веб-приложений. Однако его использование влечёт за собой проблемы с SEO. Это связано с двумя причинами. Во-первых, для доставки контента Angular в значительной мере опирается на скрипты, в результате чего некоторые поисковые роботы «не видят» тот контент, который видят пользователи. В качестве примера взглянем на [страницу](https://searchengines.guru/go?link=https://angular.io/guide/universal) с документацией по Angular Universal. Она отображается при помощи Angular. По мере прокрутки страницы вниз отображается довольно много контента. И кажется, что весь этот контент индексируется. Однако это не обязательно так. При клике правой кнопкой мыши и выборе пункта «Просмотр кода страницы» в контекстном меню браузера – будут видны лишь 100 строчек исходного кода. Но там не будет того контента, который виден пользователю, когда просматривается страница в обычном режиме. Это, если вкратце, и есть проблема с Angular. Пользователи будут видеть контент, тогда как поисковые роботы – исходный код, в котором контента нет. Есть ещё одна проблема с SEO - скорость. Приложения на Angular часто имеют низкую скорость загрузки. Некоторые сайты в течение нескольких секунд, пока не загрузится главная страница, будут отображать пустой экран. Это может привести к уменьшению количества посещений пользователями, которые не готовы терпеть такого рода задержки. Кроме того, скорость загрузки является фактором ранжирования в мобильном поиске, поэтому позиции сайта могут пострадать, если он загружается слишком медленно на мобильных устройствах.

Google утверждает, что их поисковые роботы могут индексировать управляемые скриптами сайты. Есть много фактов, подтверждающих эти заявления, но это не значит, что благодаря этому можно избежать лишней работы при оптимизации приложения Angular. Для начала, Google - это не единственная поисковая система в мире. Если требуется, чтобы приложение на Angular ранжировалось в Bing или DuckDuckGo, то потребуется предпринять ряд шагов для того, чтобы это произошло. Так же, может быть ситуация, когда поисковые роботы Google умеют индексировать некоторые сайты на Angular, но не «наши». Не все Angular приложения создаются одинаковыми. «Наше» как раз может быть исключением для алгоритма Google. Можно найти не мало историй разработки сайтов, которые требовалось перевести с HTML на Angular. В большинстве случаев они теряли огромное количество трафика из поисковых систем, который затем приходилось восстанавливать.

К счастью, есть способы сделать приложение на Angular более дружественным к SEO. Один из популярных вариантов решения этой проблемы - [динамический рендеринг.](https://searchengines.guru/ru/news/2014983) Для этого используется такой инструмент, как [Puppeteer](https://developers.google.com/web/tools/puppeteer/" \t "_blank), чтобы сгенерировать HTML-файлы, которые будет легче потреблять веб-краулерам. Затем потребуется настроить сервер таким образом, чтобы поисковые роботы направлялись на предварительно обработанные страницы, а пользователи просматривали Angular приложение в обычном режиме. Это достойное решение, но оно по-прежнему не решает проблему скорости. Для этого, более целесообразным будет использовать Angular Universal. Angular Universal - это модуль Angular, который запускает приложение на сервере, а не в браузере. И это важное различие. Обычно приложения на базе Angular являются клиентскими. Для поисковых роботов проблема состоит в том, что они не всегда «обрабатывают» код на стороне клиента, как это делают браузеры, когда предоставляют пользователю веб-страницу. Вот почему видно несоответствие между контентом страницы с документацией по Angular Universal и её исходным кодом. Angular Universal управляет рендерингом на стороне сервера (SSR). Он обрабатывает содержимое HTML и CSS, показываемое пользователю, заранее – до того, как пользователь его увидит. Это означает, что пользователь будет загружать статическую HTML-страницу вместо выполнения кода на стороне клиента. В результате страница будет загружаться быстрее. Кроме того, поскольку это статический HTML, поисковые роботы смогут индексировать контент. Если посмотреть на это с точки зрения интернет-маркетинга, то становится понятно, что борьба в этом пространстве ведётся, по большей части, за представленность в интернете. Вот почему привлекаются инфлюенсеры, публикуются обновления в социальных медиа и оптимизируются сайты для поисковых систем. Проще говоря: сайт не будет ранжироваться, если он не может быть проиндексирован. Если сайт работает на базе Angular, то нужно предпринять дополнительные шаги, чтобы его содержимое появлялось в поисковых системах. Эти проблемы как раз разрешает Angular Universal. Недостатком, конечно, является тот факт, что его внедрение займет больше времени. Для выполнения этой задачи скорее всего потребуется нанять квалифицированных программистов, чтобы они добавили SSR на сайт. Однако, эти расходы со временем окупятся, если сайт будет хорошо ранжироваться по ключевым словам, связанным с его нишей.

Для внедрения SSR в веб-приложение, требуются хотя бы базовые знания об Angular, работе с интерфейсом командной строки (CLI), TypeSrcipt и веб-серверах. В противном случае, существует риск неоправданных ожиданий. Чтобы внедрить Angular Universal, нужно выполнить следующие шаги:

* Установка зависимости;
* Обновление приложения Angular;
* Создания пакета Universal используйте CLI;
* Настройка сервера для запуска пакета Universal;
* Запуск приложения на сервере.

Далее о каждый из этих шагов подробнее.

**1. Установка зависимости.** Те у кого уже имеется опыт работы с Angular, уже знают о [Node.js](https://searchengines.guru/go?link=https://nodejs.org/en/" \t "_blank" \o "https://nodejs.org/en/). Это среда выполнения, которая компилирует код TypeScript в приложение JavaScript. Node.js поставляется с менеджером пакетов, называемым Node Package Manager или, если коротко, npm. Его нужно использовать его для инсталляции зависимостей, а именно открыть командную строку и выполните следующий код:

*npm install –save @angular/platform-server @nguniversal/module-map-ngfactory-loader ts-loader*

Потребуется несколько минут, чтобы всё установить.

**2. Обновление приложения Angular.** Далее, необходимо подготовить приложение Angular к внедрению Universal. Этот процесс включает четыре шага:

* **Добавление поддержки Universal**. Нужно открыть корневой модуль (обычно AppModule) и добавить идентификатор приложения в импорт BrowserModule. Это можно сделать в разделе «Imports» чуть ниже @NgModule.
* **Создание корневого модуля сервера**. Теперь, нужно создать новый модуль с именем AppServerModule. Не будет лишним убедиться, что он импортирует ServerModule из зависимости платформа-сервер, которая была добавлена на предыдущем шаге.
* **Создание основного файла**. Необходим основной файл для пакета Universal. Его можно создать в корне (в папке src) и экспортировать класс AppServerModule из этого файла.
* **Создание файла конфигурации**. Класс AppServerModule нуждается в файле конфигурации. Он создается в формате JSON.

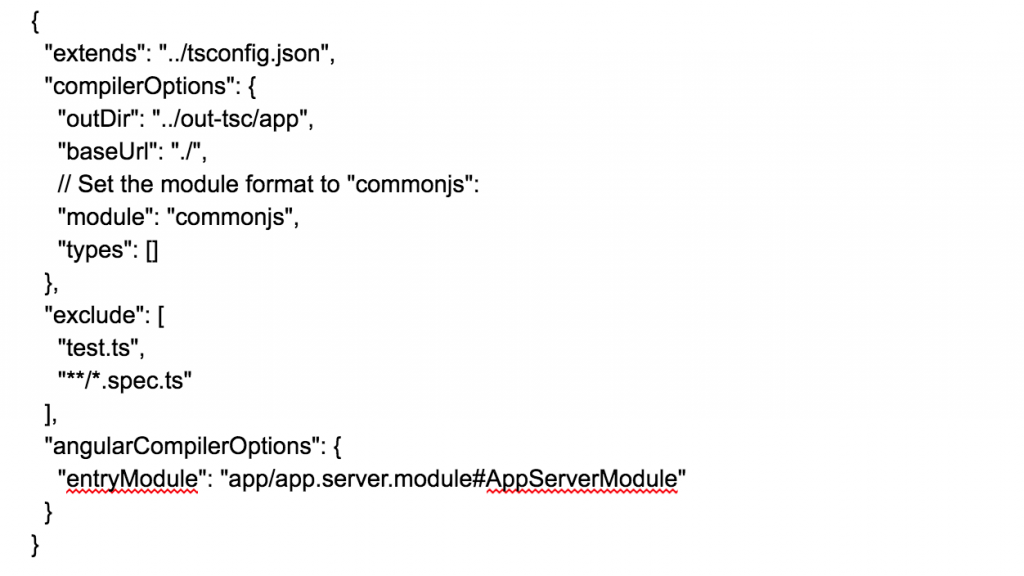


Рис. \_

По итогу должно получиться примерно следующее (рис. ^).

**3. Создание новой цели сборки.** Исходный каталог Angular должен включать файл с именем angular.json. Вам необходимо обновить этот файл в разделе «architect». Это будет выглядеть примерно так:

"architect": {  
  "build": { … }  
  "server": {  
    "builder": "@angular-devkit/build-angular:server",  
    "options": {  
      "outputPath": "dist/my-project-server",  
      "main": "src/main.server.ts",  
      "tsConfig": "src/tsconfig.server.json"  
    }  
  }  
 }

Следует обратить внимание на атрибут «builder», который находится на четвёртой строчке. Значение после двоеточия («server») - это имя сервера. Его можете изменить, если есть необходимость назвать его по-другому. Теперь можно создать своё приложение. Если сохранить сервер под именем «server», тогда нужно просто перейти в командную строку и ввести следующее:

ng run my-project:server

В результате выводится примерно следующее:

Date: 2018-12-12T12:42:09.601Z

Hash: 1caced0e9434007fd7ac

Time: 4122ms

chunk {0} main.js (main) 9.49 kB [entry] [rendered]

chunk {1} styles.css (styles) 0 bytes [entry] [rendered]

**4. Настрока сервера.** На этом этапе нужно настроить сервер Universal для запуска пакета. Это то, как будет сериализовыватся приложение и возвращать его в браузер. Для этого создаётся новый файл с именем server.ts. В этом файле определяется движок приложения.

**5. Запуск приложения на сервере.** Наконец, можно запустить приложение на сервере. Для этого нужно настроить веб-пакет, который обрабатывает файл server.ts, созданный на предыдущем шаге. Назвать файл конфигурации для удобства можно webpack.server.config.js. Рекомендуется просмотреть официальную документацию по Angular Universal, чтобы определить точный вид кода в данном файле. Возможно, придётся адаптировать этот код к вашему собственному соглашению об именах. Когда будет закончена работа с файлом, в папке dist появятся две папки: браузер и сервер. Чтобы выполнить серверный код, нужно ввести в командной строке следующее:

node dist/server.js

Хотя Angular позволяет разработчикам быстро создавать высококачественные приложения, они не всегда работает хорошо с точки зрения поисковых роботов и SEO. К счастью, Angular Universal способен предварительно рендерить страницы Angular приложений и выдавать статичный HTML. Это позволяет краулерам находить их и индексировать, а также ускоряет загрузку.

# Ленивая загрузка (Lazy-Loading)

## Общее понимание ленивой загрузки

Разделение кода это процесс, который очень явно разделяет наш код на несколько частей. Но что и как и мы делим? Разделение кода позволяет нам взять наш полный пакет и разбить его на несколько частей.В этом вся суть этого процесса, Webpack позволяет нам сделать это очень легко с загрузчиком для Angular. Наше приложение по сути становится множеством небольших приложений, которые мы обычно называем «модулями». Эти модули могут быть загружены по требованию [8].

В некоторых случаях какое-то содержание страницы нужно видеть не всем или не всегда, но на загрузку и генерацию этого содержания тратится много времени и ресурсов. В этом случае есть смысл сделать загрузку «ленивой», не загружая информацию, пока она не понадобится [9]. Так же исследования показали, что:

* 47% пользователей не хотят ждать загрузки сайта больше двух секунд;
* Посетители интернет-магазинов тоже не хотят ждать больше двух секунд, но Google намерен добиться уменьшения этого времени до половины секунды;
* 57% пользователей уйдут с сайта, если на телефоне он будет загружаться дольше трех секунд [20].

Lazy loading - так же является шаблоном проектирования в компьютерном программировании для отсрочки инициализации объекта до его вызова. Применение данной концепции очень велико на сегодняшний день и наибольшая его концентрация собралась вокруг веб-программирования. Загрузка контента по частям делает возможным реализацию, очень большого функционала веб-приложения, который может довольно быстро и стабильно работать на “клиенте” [10]. Но есть и другие примеры. Один из таких примеров это, ОРМ фреймворки, которые для удобства работы с ними разработчику, загружают данные в готовые, но пустые объекты. В последствии к ним легко обращаться средствами синтаксиса языка, а также, к их сопряжениям с другими объектами. Это в разы ускоряет разработку, однако нагружает “машину” и тратит ресурсы на задачи не требуемы в данный момент. Что бы не загружать лишние данные и экономить ресурсы используется ленивая загрузка. Это может повысить эффективность работы программы, при правильном использовании [22]. Существует четыре основных способа реализации данного шаблона проектирования:

1. Lazy initialization (Отложенная инициализация);
2. Virtual proxy (Виртуальный прокси);
3. Ghost (Призрак);
4. Value holder (Контейнер значения).

Каждый из них имеет свои плюсы и минусы.

**Lazy initialization (Отложенная инициализация) -** при данной реализации загружаемый объект изначально имеет значение null, при каждом запросе объекта проверяется его значение на наличие null и создает его «на лету», прежде чем возвращать его первым.

Пример кода на языке C#:

private int myWidgetID;

private Widget myWidget = null;

public Widget MyWidget

{

get

{

if (myWidget == null)

{

myWidget = Widget.Load(myWidgetID);

}

return myWidget;

}

}

Этот метод наиболее прост в реализации.

**Virtual proxy (Виртуальный прокси) -** эта реализация построена как объект с тем же интерфейсом, что и у реального объекта. При первом вызове одного из его методов он загружает реальный объект, а затем делегирует.

**Ghost (Призрак) -** данная реализация, это объект без каких-либо данных, который будет загружен в частичном состоянии. Он может содержать только идентификатор объекта, но загружает свои собственные данные при первом обращении к одному из его свойств. Например, предположим, что пользователь собирается запросить контент через онлайн-форму. Все, что мы знаем во время создания, это то, что к контенту будет доступ, но какое действие или контент неизвестен.

**Value holder (Контейнер значения) -** это общий объект, который обрабатывает ленивое поведение загрузки, и появляется на месте полей данных объекта. Обычно объект с методом getValue, который вызывает загрузку.

Пример кода на языке C#:

private  ValueHolder < Widget >  valueHolder ;

public  Widget  MyWidget

{

get

{

return  valueHolder . GetValue ();

}

}

Клиент вызывает метод getValue.

В Angular уже есть программный API для ленивой загрузки - NgModule. В Angular CLI он имеет прямую зависимость от базовой инструментальной привязки webpack для разделения блоков и ленивой загрузки. Это значит, чтобы реализовать технологию Lazy-Loading в приложении Angular нам нужно выполнить 3 пункта, а именно [1]:

1. Создать функциональный модуль;
2. Создать модуль маршрутизации модуля функций;
3. Настроить маршруты.

Для компонентов, которые требуется загружать с помощью ленивой загрузки мы создаём функциональные модули. Для этого зайдем в папку с проектом через, командную строку, и после этого введем команду для генерации функционального модуля: ng generate module [имя модуля] (рис. 2.1).

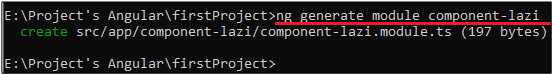


Рис. 2.1. Создаем модуль с помощью Angular CLI в командной строке.

После этого в нашем проекте мы увидим новую папку и наш модуль в ней (рис 2.2).

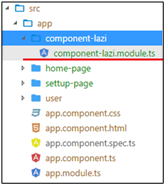


Рис. 2.2. Созданный модуль.

Теперь в этот модуль (component-lazi.module.ts) нам нужно добавить новые маршруты, и декларировать компоненты.

import { HomePageComponent } from './home-page.component';   
import { RouterModule, Routes } from '@angular/router';

const routes : Routes = [   
{   
path : ' ',   
component : HomePageComponent   
}   
];

@NgModule({   
imports : [   
CommonModule,   
RouterModule.forChild(routes)   
],   
declarations : [ HomePageComponent ],   
exports : [ RouterModule ]   
})   
export class ComponentLazyModule { }

И так мы импортировали, а после задекларировали HomePageComponent, который будет загружаться с помощью ленивой загрузки. Если до этого он был добавлен в корневой модуль, то теперь его следует оттуда удалить. После мы так же импортировали модуль маршрутизации. Следующим шагом будет настройка корневого модуля и корневого маршрута.

const routes : Routes = [   
{   
path : 'lazi-loading ',   
loadChildren : 'app/component-lazy/component-' + 'lazy.module#ComponentLazyModule'   
}   
];

@NgModule({   
imports : [   
CommonModule,   
RouterModule.forChild(routes)   
],   
declarations : [ HomePageComponent ],   
exports : [ RouterModule ]   
})   
export class ComponentLaziModule { }

loadChildren сопровождается строкой, являющейся путем к модулю, хэш-меткой или #именем класса модуля.

Мы можем проверить, загружается ли модуль, с помощью инструментов разработчика Chrome. В Chrome откройте инструменты разработчика, нажав Cmd+Option+i на Mac или Ctrl+Alt+i на ПК и перейдя на вкладку «Network» (рис. 2.3) [1].

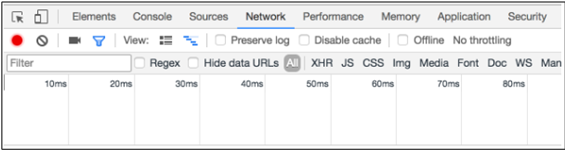


Рис. 2.3. DevTools в браузере Google Chrome.

Предварительно очистим то что записал браузер (рис. 2.4) во время загрузки нашего приложения.

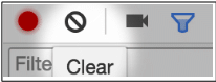


Рис. 2.4. Кнопка Clear в DevTools.

Теперь нужно спровоцировать загрузку нашего компонента путем маршрутизации. После этого мы увидим новую запись (рис 2.5).

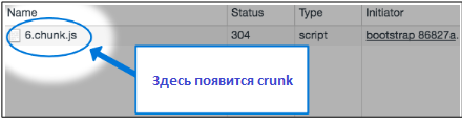


Рис. 2.5. Отслеживается загруженый файл.

Можно заметить, что RouterModule.forRoot(routes) был добавлен к app-routing.module.ts imports массиву. Это говорит Angular, что данный модуль AppRoutingModule является модулем маршрутизации и forRoot() указывает, что это корневой модуль маршрутизации[2]. Он настраивает все маршруты, которые мы передаем ему, дает вам доступ к директивам маршрутизатора и регистрирует RouterService. Использовать forRoot() можно лишь один раз в приложении в AppRoutingModule на корневом уровне. Также мы добавили RouterModule.forChild(routes) модули маршрутизации функций. Таким образом, Angular знает, что этот список маршрутов отвечает только за предоставление дополнительных маршрутов и предназначен для функциональных модулей. Можно использовать forChild() в нескольких модулях. forRoot() содержит конфигурацию инжектора, которая является глобальной; таких как настройка маршрутизатора [17]. forChild() не имеет конфигурации инжектора, только директивы, такие как RouterOutlet и RouterLink.

## Ленивая загрузка модулей в Angular

Angular это фреймворк для создания клиентских веб приложений на HTML и TypeScript. Он реализует основные и дополнительные функции для приложения представляя их библиотеками на языке TypeScript, которые мы можем импортировать в модули [1].

import { NgModule } from '@angular/core';

Основными строительными блоками Angular приложения являются модули, которые предоставляют контекст компиляции для компонентов. Директива NgModules собирает связанный код в функциональные множества. Angular приложение определяется набором таких директив [2].

@NgModule({imports : [ BrowserModule ],

providers : [ Logger ],

declarations : [ AppComponent ],

exports : [ AppComponent ],

bootstrap : [ AppComponent ]})

export class AppModule {}

У приложения всегда есть корневой модуль, который определяет загрузку приложения и обычно содержит много функциональных модулей. Компоненты содержат представления, которые представляют собой набор HTML элементов, которые Angular может изменять в соответствии с нашей программной логикой и данными. У каждого приложения есть хотя бы один корневой компонент (в только что созданном проекте им по умолчанию будет AppComponent).

@Component({   
selector : 'app-root',   
templateUrl : './app.component.html',   
styleUrls : [ './app.component.css' ]   
})   
export class AppComponent

Компоненты используют сервисы, которые обеспечивают определенную функциональность, не связанную напрямую с представлениями [12]. Сервисы (они же провайдеры) могут быть внедрены в компоненты в качестве зависимостей, что делает наш код модульным, многоразовым и эффективным. Компонента и сервисы - это простые классы, с декораторами, которые определяют их тип и предоставляют метаданные, которые сообщают Angular, как их использовать [1]. Метаданные класса компонента связывают его с шаблоном (свойство templateUrl) определяющим представление. Шаблон объединяет обычный HTML с Angular директивами, которые позволяют Angular изменять HTML-код перед его отображением. Метаданные класса сервиса предоставляют информацию, необходимую Angular, чтобы сделать ее доступной для компонентов через Injection Dependency (DI) [3]. Компоненты приложения обычно определяют много представлений, упорядоченных иерархически. В Angular есть Router сервис, который призван помочь нам назначить пути навигации между ними [14]. Маршрутизатор предоставляет сложные навигационные возможности в браузере. Шаблон сочетает HTML с привязкой Angular к разметке с помощью директив, которые могут изменять элементы HTML до их отображения.

<p myDirective> using my directive! <p>

Директивы шаблонов предоставляют программную логику, а привязка к разметке связывает данные приложения и объектную модель документа (DOM). Связывание событий позволяет вашему приложению реагировать на ввод пользователя в целевой среде путем обновления данных приложения [3]. Связывание свойств позволяет вам интерполировать значения, которые вычисляются из ваших данных приложения в HTML. Перед отображением представления, Angular оценивает директивы и разрешает синтаксис привязки в шаблоне для изменения элементов HTML и DOM в соответствии с вашими программными данными и логикой. Angular поддерживает двустороннюю привязку данных. Это означает, что изменения в DOM, такие как выбор пользователя, также могут быть отражены обратно в данные программы [2]. Шаблоны также могут использовать pipe(каналы), чтобы улучшить работу пользователя, изменив значения для отображения. Используются pipe для отображения, например, дат меняя формат с '02 февраль 2018 09:25:18:123' на '02.02.18 09:25' или наоборот, или для других похожих операций с любыми данными [2].

<p> using pipe for date {{ myDate | date }}</p>

Angular уже имеет pipe'ы для общих преобразований, но также можно определить свои собственные. Для данных или логики, которые не связаны с определенным представлением и должны быть использованы в разных компонентах, обычно создается сервис.

constructor (private service: MyService){}

Для определения сервиса есть специальный декоратор @Injectable.  Данный декоратор содержит метаданные, которые позволяют нашему сервису внедряться в клиентские компоненты в качестве зависимости.

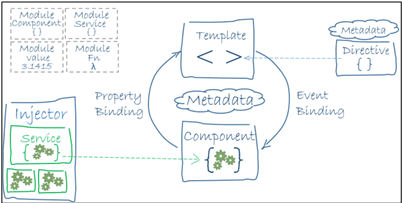


Рис. 1.1. Диаграмма показывает, как эти основные части связаны.

Dependency injection (или DI) позволяет сохранить класс компонентов и повысить эффективность (рис. 1.1). Компоненты не извлекают данные с сервера и не проверяют вход пользователя, они перекладывают эти задачи на сервисы.

Благодаря языку TypeScript в Angular есть возможность полноценного использования ООП (Объектно-Ориентированное Программирование). И конечно же мы можем создавать классы наделяя их определенным функционалом. Но на самом деле классы в Angular играют куда большую роль. Angular, кроме объектно-ориентированной парадигмы использует так же декларативное программирование. Декларативное программирование - это парадигма программирования, в которой задаётся спецификация решения задачи, то есть описывается, что представляет собой проблема и ожидаемый результат [3]. Мы можем интерпретировать это примерно так: наши классы будут задекларированы для того, чтобы Angular раздал им свои роли. Это делается с помощью декораторов и вот некоторые из них [1]:

* @NgModule - этот декоратор говорит Angular, что данный класс является модулем, который нужен для определения сервисов, компонентов, маршрутов, пайпов и тд. С помощью этого декоратора происходит разделение приложения на модули;
* @Component - этот декоратор говорит Angular, что класс является компонентом. Компоненты нужны для отображения и логических операций над данными;
* @Injectable - этот декоратор говорит Angular, что класс является сервисом. Сервисы нужны для хранения и использования общим данных которые должны быть доступны нескольким компонентом (Например, данные о пользователе);
* @Pipe - этот декоратор говорит Angular, что класс является фильтром. Фильтры используются для динамического изменения данных или формата данных внутри HTML шаблона;
* @Directive - этот декоратор говорит Angular, что класс является директивой. Директивы нужны для того что производить манипуляции над HTML шаблоном.

После этой процедуры классы обретут гораздо большие возможности. Декораторы могут быть использованы для аннотации класса, свойства, метода или параметра. Давайте подробно рассмотрим каждый из этих типов.

Декомпозиция - разделение целого на части. Часто в веб приложениях существует необходимость обратиться к компоненту по ссылке [25]. И SPA не являются исключением. В Angular есть Router, который отвечает за навигацию приложения. Он позволяет через Url переходить с одного компонента на другой [1].

У браузера имеется модель навигации приложения:

1. Если ввести URL-адрес в адресной строке, браузер перейдет на соответствующую страницу;
2. Нажимая ссылку на странице, браузер перейдет на новую страницу;
3. Используя кнопки браузера назад и вперед, браузер переместится назад и вперед по истории страниц, которые открывались.

Router в Angular заимствует эту модель навигации. Он может интерпретировать URL-адрес браузера в качестве инструкции для перехода к представлению, создаваемому клиентом. Он может передавать необязательные параметры вместе с компонентом. Мы можем привязать маршрутизатор к ссылкам на странице, и по нажатию на них он перейдет к соответствующему компоненту приложения. Маршрутизатор регистрирует активность в журнале истории браузера, так что кнопки назад и вперед будут работать [2].

Для добавления маршрутизации в наше веб приложение нужно, чтобы в файле index.html в качестве первого дочернего элемента тега <head> был добавлен элемент <base>.

<base href="/">

Это нужно для того, чтобы сообщить маршрутизатору, как составлять URL-адреса навигации.

Angular Router - не является обязательным условием для проектирования приложения, но он позволяет закрепить за конкретным компонентом собственный URL-адрес. Это не часть ядра Angular. Он находится в пакете библиотеки @angular/router. Импортируйте то, что вам нужно из данного пакета, как из любого другого пакета Angular.

import { RouterModule, Routes } from '@angular/router';

Маршрутизированное Angular приложение содержит один экземпляр Router сервиса. Когда URL-адрес браузера изменяется, маршрутизатор ищет соответствующий, Route из которого он может определить отображаемый компонент. Маршрутизатор не имеет маршрутов, пока его не настроить. Настраивается маршрутизатор с помощью RouterModule.forRoot метода и добавляется в AppModule, в декоратор NgModule, в свойство imports [14].

const appRoutes : Routes = [   
{   
path : 'crisis-center',   
component : CrisisListComponent   
},   
{   
path : 'heroes',   
component : HeroListComponent   
data : { title: 'Heroes List' }   
},   
{   
path : ' ',   
redirectTo : '/heroes',   
pathMatch : 'full'   
},   
{   
path: '\*\*',   
component: PageNotFoundComponent   
}   
];

Созданный appRoutes это массив маршрутов, который описывает перемещения по Url - адресам. :id во втором маршруте является маркером для параметра маршрута. В URL-адресе, таком как /hero/42, "42" будет значением id параметра. Каждый из маршрутов сопоставляется с URL-адресом (в свойстве path) компонента [1]. Маршрутизатор анализирует и создает окончательный URL-адрес, позволяя использовать как относительные, так и абсолютные пути при навигации между видами приложений. Теперь его нужно передать в RouterModule.forRoo метод в модуле, в свойство imports для настройки маршрутизатора.

@NgModule({   
imports : [   
RouterModule.forRoot(   
appRoutes,   
{ enableTracing : true}   
)   
],   
...}) export class AppModule { }

Если нам нужно посмотреть, какие события происходят во время жизненного цикла навигации, есть опция enableTracing. При значении true выводится каждое событие маршрутизатора, которое будет происходить в течение каждого жизненного цикла навигации в консоль браузера. Эту опцию следует использовать только для отладки.  Устанавливаете enableTracing: true опцию в объекте, переданном в качестве второго аргумента RouterModule.forRoot() методу (так же как это показыно на картинке выше) [1].

Когда URL-адрес браузера для нашего приложения становится /heroes, маршрутизатор сопоставляет этот URL-адрес с маршрутом /heroes и отображает HeroListComponent внутри тега <router-outlet> который размещён в HTML-представлении хоста.

<router-outlet></router-outlet>

URL-адрес может поступать непосредственно из адресной строки браузера. Но большую часть времени перемещение происходит в результате какого-либо пользовательского действия, например, щелчка мыши по кнопке.

<h1>Angular Router</h1>   
<nav>   
<a routerLink="/crisis-center" routerLinkActive="active">   
Crisis Center   
<a>   
<a routerLink="/heroes" routerLinkActive="active">   
Heroes   
<a>   
</nav>   
<router-outlet></router-outlet>

В директивах routerLink по тегам предоставляется контроль над маршрутизатором этих элементов. Директивы routerLinkActive меняют стиль отображения элемента если маршрут элемента в данный момент активен.

Так же у маршрутов имеется защита, называемая Guard. Guard может разрешить или запретить навигацию по определенному маршруту. К примеру, для доступа к определенному компоненту требуется наличие каких-то условий, отталкиваясь от которых мы можем предоставить или не предоставить пользователю доступ. Guards защищают доступ к маршруту. Один из типов защиты CanActivate позволит нам управлять доступом к компоненту при маршрутизации. Для того, чтобы его реализовать создадим файл about.guard.ts в папке src/app.

import {    
CanActivate,    
ActivatedRouteSnapshot,   
RouterStateSnapshot    
} from '@angular/router';   
import { Observable } from 'rxjs';

export class AboutGuard implements CanActivate {   
canActivate (route : ActivatedRouteSnapshot, state :  RouterStateSnapshot) : Observable<boolean> | boolean {    
return true;   
}   
}

Класс AboutGuard реализует интерфейс CanActivate. Mетод CanActivate() принимает параметры - ActivatedRouteSnapshot и RouterStateSnapshot. Они содержат информацию о запросе [3]. ActivatedRouteSnapshot позволяет получить различную информацию из запроса, такую как, например, параметры маршрута и строки запроса. Если CanActivate() вернет true это будет означать что переход по маршруту одобрен и маршрутизация сработает.

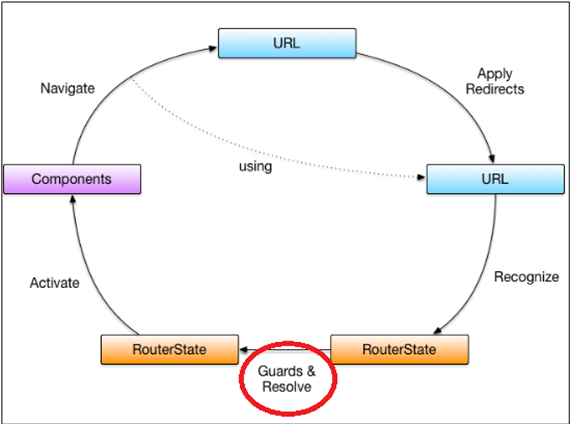


Рис. 1.2. Жизненный цикл маршрутизации.

Guard проверяется раньше, чем загружается компонент (рис. 1.2).

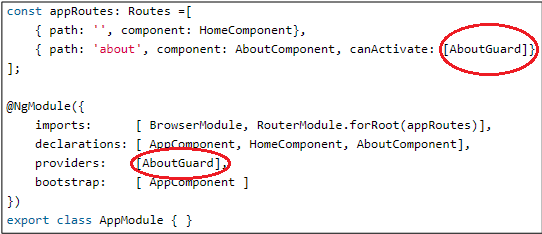


Рис. 1.3. Добавляем в свойство canActivate и providers Guard.

Чтобы использовать Guard нужно добавить созданный класс AboutGuard в корневой модуль приложения в свойство providers, а также добавить свойство canActivate и в него так же добавить класс AboutGuard (рис. 1.3) [2].

Безусловно Angular не единственное SPA решение для вашего веб приложения. Тем не менее его рейтинг ежегодно растет во всевозможных топах и сравнениях с другими инструментами.

*Преимущества:*

* Поддержка вэб-компонентов. Весь код делится на компоненты, сервисы и мо

дули. Это позволяет разработчику быстро найти нужный участок кода и начать работать в нём. Так как компоненты разделены изменение кода в одном ни коим образом не повлияет на другой;

* Использование TypeScript. Самое большое коммерческое преимущество TypeScript состоит в его инструментарии. Этот язык даёт возможности современного автозаполнения, навигации и рефакторинга. Такие инструменты становятся практически незаменимыми при работе с большими проектами;
* Отличная производительность. Angular не проводит глубокий сравнительный анализ объектов. Если какой-то элемент добавить в массив данных, изменение пути не будет обнаружено. Это касается и свойств объекта, пока они не связаны напрямую с View;
* Angular CLI. CLI фреймворка Angular даёт возможность с лёгкостью создавать приложение, которое работает по умолчанию. Это соответствует лучшим современным тенденциям.

*Недостатки:*

* Тяжело освоить. Это ни на что не похожий полноценный фреймворк с уже существующими в нём реактивными привязками к шаблонам. Это не игрушка для новичка, это профессиональный и очень мощный инструмент для разработки больших проектов.

Всё чаще для разработки своих веб приложений выбирают Angular. По какой причине это происходит? Главная причина тому усердный труд разработчиков Angular. У данного фреймворка за спиной поддержка огромной компания, которая очень редко дает усомниться в качестве своих разработок. За последние года Angular стал одним из самых популярных фреймфорков в мире front-end разработки. По этой причине многие производители сторонних сервисов (библиотек, инструментов) разрабатывают так же совместимые с Angular версии своих продуктов. Некоторые из них Angular уже использует внутри своей архитектуры (например, RxJS библиотеку) и загружает по умолчанию при создании нового проекта (но не импортирует их), другие сервисы нужно самостоятельно искать и устанавливать. Как правило местом для хранения этих сервисов вполне обоснованно является облачный сервер npm. Npm это пакетный менеджер, использующийся для скачивания или загрузки на сервер пакетов (пакетом в Node.js называется один или несколько JavaScript-файлов, представляющих собой какую-то библиотеку или инструмент), обладающий самой большой пакетной экосистемой в мире с открытым исходным кодом [3]. Именно с помощью npm мы, кстати говоря, устанавливает на свой пк Angular CLI и даже при создании нового проект неявно его используем. Npm уже входит в состав программы Node.js поэтому не требует отдельной установки.

В качестве примера попробуем установить Material Angular - пакет для, создания элементов в стиле Material Design. Для этого откроем терминал своего редактора кода, который мы используем для разработки Angular приложения, либо воспользуемся командной строкой операционной системы и предварительно зайдя в корневую папку проекта пропишем там (рис. 3.4):

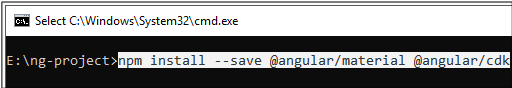


Рис. 3.4. Установка Angular Material и Angular CDK.

Этот шаг установит Angular Material и Angular CDK. Следующий шаг установит модуль с анимациями (рис. 3.5) так как некоторые компоненты уже содержат эти анимации.

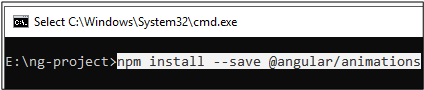


Рис. 3.5. Установит модуля с анимациями.

После этого консоль нам, больше не нужна и мы можем импортировать файлы в проект. Мы будем делать это в нашем главном модуле AppModule, однако если Angular компонент, в котором будет использоваться содержимое пакета Material Angular, импортирован в какой-нибудь дочерний модуль, то имеет смысл для лучшей оптимизации приложения импортировать нужное содержимое пакета именно в этот дочерний модуль. Первым делом импортируем анимации (его следует импортировать только в одном, в главном, модуле):

import { BrowserAnimationsModule } from '@angular/platform- browser/animations';

@NgModule({   
imports: [   
BrowserAnimationsModule   
]   
})   
export class AppModule { }

Далее импортируем нужные компоненты из пакета. Обратите внимание на, то что мы импортируем только те компоненты и пакета, которые собираемся использовать. Тем самым при компиляции наше приложение будет содержать только используемые компоненты. Это уменьшит её размеры и на выходе мы получим более быструю загрузку приложения.

import {    
MatButtonModule,   
MatToolbarModule   
} from '@angular/material';

@NgModule({   
imports: [   
BrowserAnimationsModule,   
MatButtonModule,   
MatToolbarModule   
]   
})   
export class AppModule { }

Последнее что мы делаем это добавляем в главный styles.css файл (рис. 3.6), строку для импортирования темы (рис. 3.7), которая уже задана для компонентов этого пакета. Мы так же может создавать свои темы.

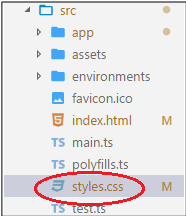


Рис. 3.6. Находим файл styles.css.

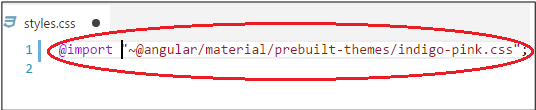


Рис. 3.7. Импортируем тему.

Теперь мы можем использовать пакет Material Angular для создания своих приложений. Он подключен и готов к работе. Остальные пакеты, которые могут нам понадобиться устанавливаются не сложнее чем этот так как каждый производитель в обязательном порядке выкладывает:

1. Подробную инструкцию для установки пакета (Started);
2. Полную документацию;
3. Примеры с использования, которые кроме официального сайта выкладываются чаще на [https://github.com](https://github.com/" \t "_blank) доступные для скачивания и чуть реже на [https://stackblitz.com](https://stackblitz.com/" \t "_blank) доступные в онлайн режиме.

Для более детального описание курсовых работ воспользуемся одним из WYSIWYG editor. WYSIWYG - это текстовые редакторы конвертирующие всё написанное в HTML код, а сохранённые картинки в byte. Мы будем использовать Quill. Quill больше всего соответствует дизайну нашего приложения. После установки получился полноценный текстовый редактор в веб приложении (рис 3.8).



Рис. 3.8. Панель с функциями редактора.

Quill очень гибкий и настраивается легко под любые нужды будь то запись математических формул или же статья с красочным ярким заголовком с соблюдением всех правил форматирования текста [21].

Так как авторизация в приложении осуществляется через Google аккаунт то понадобится сервис Google Oauth 2.0. Это Api, которое позволит авторизовываться в приложении через аккаунт Google. Для начала нужно зарегистрировать приложение в Google Developers Console ([https://console.developers.google.com](https://console.developers.google.com/)) (рис. 3.9). [4]

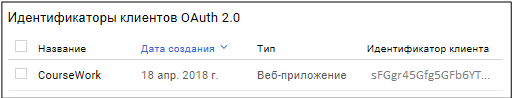


Рис. 3.9. Добавление приложения в Google Developers Console.

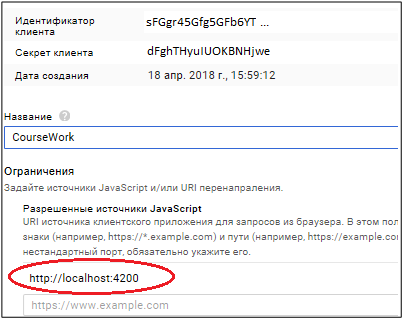


Рис. 3.10. Настройки Google Developers Console.

Для приложения создается идентификатор. В настройках нужно добавить адрес приложения (рис 3.10).

Теперь нужно добавить новый фунционал в приложение. После того как npm успешно устанавливает этот сервис в приложения нужно добавить его в корневой модуль приложения[4]:

import {   
SocialLoginModule,   
AuthServiceConfig,   
GoogleLoginProvider   
} from 'angular5-social-login';

Ввожу полученый идентификатор:

export function getAuthServiceConfigs() {   
const config = new AuthServiceConfig(   
[{   
id: GoogleLoginProvider.PROVIDER\_ID,   
provider: new GoogleLoginProvider('496766590....')   
}]);   
return config;   
}

Добавляю модули и сервисы в проложения:

@NgModule({   
imports: [   
SocialLoginModule   
],   
providers: [   
{   
provide: AuthServiceConfig,   
useFactory: getAuthServiceConfigs   
}   
]   
export class AppModule { }

Для авторизации в приложении появляется стандартное окошко от Google. После успешной авторизации данные сравниваются уже с нашей базой данных и приложение открывает доступ пользователю. Если с устройства уже был совершен вход в аккаунт Google то приложение автоматически будет проверять сохранённые в браузере данные и соответственно автоматически авторизовывать пользователя.

В Angular так же присутствуют встроенные механизмы для реализации реактивного программирования. Если говорить простыми словами, не вдаваясь в технические детали, то преимущество реактивного программирования состоит в том, чтобы мгновенно узнавать о любом изменении в переменной и как-то на это реагировать уже с помощью кода. Это полезно тем, что не приходится дублировать много раз код и придумывать непонятные, плохо читаемые алгоритмы для простой замены значений, в одном или нескольких местах кода [24]. Одна из современных библиотек реактивного программирования носит название RxJs [7]. Данная библиотека может быть легко импортирована в приложение Angular, однако и сам фреймворм в своей внутренней архитектуре содержит некоторые её реализации для мгновенного реагирования на изменения. Это легко заметно на примере работы односторонней и двусторонней привязки шаблонов компонентов (когда значение переменной меняется, шаблон компонента на странице перерисовывается) и не только. HttpClient для отправки асинхронных запросов так же использует библиотеку RxJs [16].

HttpClient.get(this.url).subscribe(data => { // …  });

Существуют наблюдатели, они обеспечивают поддержку для передачи сообщений между издателями и подписчиками в нашем приложении. Наблюдения обеспечивают значительные преимущества по сравнению с другими методами обработки событий, асинхронного программирования и обработки нескольких значений. Так же существуют наблюдаемые, которые являются декларативными. Мы определяем функцию для публикации значений, но она не выполняется до тех пор, пока потребитель не подпишется на неё. Затем подписчик или подписчики получают уведомления до завершения функции или до тех пор, пока они не отпишутся [7]. Наблюдаемый может передавать несколько значений любых типов литералов, сообщений или событий в зависимости от контекста. API для получения значений одинаковый, независимо от того, доставляются ли значения синхронно или асинхронно. Поскольку логика установки и разрыва обрабатывается наблюдаемым, в нашем коде приложения мы должны беспокоиться только о том, чтобы подписываться на потребление значений, а когда это будет сделано, отказаться от подписки. Независимо от того, был ли поток нажатием клавиши, ответом HTTP или интервальным таймером, интерфейс для прослушивания значений и остановки прослушивания одинаковый [7]. Благодаря этим преимуществам наблюдаемые(Observables) широко используются в пределах Angular и также рекомендуются для разработки приложений. Мы создаем Observable экземпляр, который определяет функцию абонента. Это функция, которая выполняется, когда потребитель вызывает метод subscribe() [1]. Это объект JavaScript, который определяет обработчики для получаемых уведомлений. subscribe() Вызов возвращает Subscription объект, который так же содержит метод unsubscribe(), предназначенный для отписки от получения уведомлений о наблюдаемом объекте.

Observer обеспечивает обработчик для получения наблюдаемых уведомлений. Это объект, который определяет методы обратного вызова для обработки трех типов уведомлений, которые может отправлять наблюдаемый [15]:

* *Next*- обработчик для каждого поставленного значения. Будет вызываться при изменениях в наблюдаемом объекте. Обязательный;
* *Error*- обработчик для уведомления об ошибке. Ошибка останавливает выполнение наблюдаемого экземпляра. Необязательный;
* *Comlete*- обработчик для уведомления об исполнении. Необязательный.

Для некоторых однотипных компонентов было бы более правильным использовать реактивное программирование. Например, компонент progress bar (рис. 3.18) сервиса Material Angular, дабы не добавлять progress bar во все компоненты Angular и не загромождать исходный код.



Рис. 3.18. Progress Bar.

Вместо этого добавим его лишь в корневой компонент AppComponent и с помощью библиотеки RxJS мы сможем "включать" и "выключать" анимацию загрузки. Создадим сервис для хранения всех реактивных объектов. Из данной библиотеки для этих целей лучше всего подойдет класс BehaviorSubjec:

import { BehaviorSubject } from 'rxjs/BehaviorSubject';

С его помощью создадим объект содержащий значение типа boolean, но являющийся реактивным:

isLoad =

new BehaviorSubject<boolean>(false);

И последнее что следует сделать это добавить в этот сервис метод, который меняет значение в реактивном объекте:

isLoading(chenged: boolean) {   
this.isLoad.next(chenged);}

После этого без труда можно подписать компонент progress bar на этот объект:

@Component({   
selector : 'app-loading',   
template : '<mat-progress-bar ' +   
'color="accent" ' +   
'\*ngIf="isLoading" ' +    
'mode="indeterminate"></mat-progress-bar>',   
})   
export class AppLoadingComponent implements OnInit {   
title = 'app';   
isLoading: boolean;   
constructor(private rx: ReactiveChangedService) {   
this.rx.isLoad.subscribe(data => this.isLoading = data);   
}   
}

А в других компонентах меняя его значение на true "включать", а на false "выключать" индикатор загрузки:

constructor (public rx: ReactiveChangedService) { }

singIn(user: any) {   
this.rx.isLoading(true);   
}

Этот принцип используется для всех обменов данными между компонентами разве, что был заменен тип данных с boolean на тот, который больше подходит каждому из них.

## Внедрение ленивой загрузки

Приложение Angular состоит из модулей. Обычно, приложение состоит из нескольких модулей, но как минимум оно имеет один корневой модуль, который по умочанию называется AppModule. Модулю необходим ряд библиотек, поэтому в самом начале файла нужно их подключить. Имена библиотек Angular начинаются с префикса @angular. Библиотеки импортируются с помощью директивы import, а устанавливаются обычно через пакетный менеджер npm. Главный модуль в приложении Angular используется как точка входа. При создании нового проекта через команду в командной строке ng new project-name генерируется такой AppModule. Как аргумент в декораторе @NgModule должен использоваться JavaScript объект. @NgModule - это декоратор, который принимает объект, свойства которого описывают метаданные модуля. Наиболее важные свойства:

* *declarations:*классы представлений (view classes), которые принадлежат модулю. Angular имеет три типа классов представлений: компоненты (components), директивы (directives), каналы (pipes);
* *exports:*набор классов представлений, которые должны использоваться в шаблонах компонентов из других модулей;
* *imports***:**другие модули, классы которых необходимы для шаблонов компонентов из текущего модуля;
* *providers:* классы, создающие сервисы, используемые модулем;
* *bootstrap:*корневой компонент, который вызывается по умолчанию при загрузке приложения.

Директива @NgModule определяет класс, как модуль. Такие классы помогают разбивать приложение на части (модули), которые взаимодействуют между собой представляя в конечном итоге целостное приложение. Другими словами, модуль - это упаковка части функционала приложения. Если нужно реализовать lazy loading в приложении, то необходимо использовать концепцию Angular Modules при проектировании приложения [3]. В свойстве declarations объявляются компоненты, которые содержатся в этом модуле. В данном случае это компонент AppComponent. Компонентов может быть несколько, они объявляются в этих же квадратных скобках через запятую. Допустим, у нашего компонента есть селектор my-app. Когда в шаблоне HTML мы пишем <my-app></my-app> приложение загружает компонент и тот HTML код, который есть в нём. AppModule говорит браузеру, о том, что нужно встроить в DOM компонент AppComponent. AppModule импортирует служебные модули Angular, такие как BrowserModule, который отвечает за работу приложения в браузере. Это сервис, который взаимодействует с нашим кодом и API браузера. Так же BrowserModule включает в себя директивы NgIf и NgFor, которые мы можем использовать в шаблонах компонентов [2]. AppModule связывает данные, их представление и браузер. В приложении мы создаем компоненты, директивы и пайпы. Чтобы сообщить нашему приложению какой функционал мы хотим добавить, нам нужно добавить их через запятую в квадратные скобки в свойства declarations объекта, который является аргументом декоратора @NgModule. После того как компоненты задекларированы можно использовать их внутри других компонентов через их селектор, который указывается в описании компонента. Предположим в нашем приложении есть сервисы, которые мы хотим использовать в нескольких компонентах. Что бы предоставить всем компонентам доступ к сервису необходимо в свойство providers добавить сервисы которые должны видеть все компоненты [2]. В Angular добавление зависимостей реализовано с помощью паттерна Dependency Incetion. Когда мы добавим наш сервис в качестве провайдера в корневой модуль AppModule, он будет доступен в любой части приложения. Наглядным примером провайдера, который нужен во всём нашем приложении можно представить сервис с данными пользователя, назовем его UserService. Его лучше всего создать в папке services (рис 3.2).

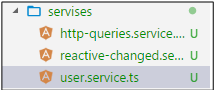


Рис. 3.2. Сервис UserService в папке с сервисами.

*Создаем сервис:*

import { Injectable } from '@angular/core';   
import { UserInfo } from '../models/models-classes';

@Injectable()   
export class UserService {   
userInfo: UserInfo;   
usersAccess = [   
'Admin',   
'Professor',   
'Student'   
];   
}

*Добавляем сервис в провайдеры (providers):*

@NgModule({   
...   
providers: [   
UserService   
]   
...   
})   
export class AppModule { }

*Используем сервисы в любом компоненте приложения:*

import { UserService } from '../servises/user.service';   
@Component({   
selector : 'app-student',   
templateUrl : './student.component.html',   
styleUrls : ['./student.component.css']   
})   
export class StudentComponent implements OnInit {   
constructor (public user : UserService) { }   
}

Провайдеры это одна из самых интересных сложных концепций модульной системы Angular. Основное правило в использовании провайдеров (сервисов) - создать экземпляр сервиса в корневом модуле и передавать по запросу этот экземпляр в другие части (компоненты) приложения.

Если представить, например, приложении с 3-мя типами пользователей, то механизм ленивой загрузки в нем состоит из 3-х дополнительных модулей. Первый модуль AppModule корневой он загружается первым и дает старт приложению. Исходя из того, что в приложении 3 типа доступа пользователей, которые не должны видеть компоненты друг друга. Решено реализовать 3 модуля для каждого типа доступа [18].

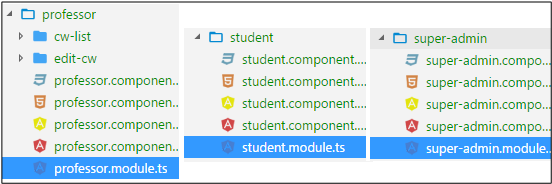


Рис. 3.3. Созданные модули в папках с компонентами.

После создания модулей (рис. 3.3) в них были добавлены компоненты, к которым они должны быть привязаны (на примере модуля student):

import { RouterModule, Routes } from '@angular/router';

const routes: Routes = [   
{   
path : ' ',   
component : StudentComponent   
}   
];

@NgModule({   
declarations: [   
StudentComponent,   
],   
imports: [   
RouterModule.forChild(routes)   
],   
exports: [RouterModule]   
})   
export class StudentModule { }

Затем были настроены маршруты в главном модуле и добавлены guard для каждого маршрута:

const routes: Routes = [   
{   
path: 'student',   
loadChildren: 'app/student/student.module#StudentModule',   
canActivate: [StudentGuard]   
},   
{   
path: 'super-admin',   
loadChildren: 'app/super-admin/super-    admin.module#SuperAdminModule',   
canActivate: [SuperAdminGuard]   
},   
{   
path: 'professor',   
loadChildren:'app/professor/professor.module#ProfessorModule',   
canActivate: [ProfessorGuard]   
}];

@NgModule({   
imports: [   
RouterModule.forRoot(routes)   
],   
exports: [RouterModule],   
providers: [   
ProfessorGuard,   
StudentGuard,   
SuperAdminGuard   
]   
})   
export class AppRoutingModule { }

Таким образом был реализован механизм ленивой загрузки в приложении.

# Angular Elements или Веб-компоненты на Angular

## Общее понимание Веб-компонентов

Работа веб-разработчиков заключается в том, чтобы выбирать правильные инструменты для любого конкретного проекта. Это может быть сложно, если важны не только насущные потребности проекта. И нужды всей команды, и если проект является частью более крупной экосистемы в компании, и как он будет поддерживаться, и как долго это нужно будет поддерживать, этот список можно ещё продолжать и продолжать. Веб-сообщество поставляет поразительное количество инструментов, библиотек и фреймворков. Порой бывает трудно успевать за всеми из них, ведь их настолько много, что этот феномен был темой разговоров в течение некоторого времени и даже сейчас. Внедрение этих новых инструментов происходит молниеносно. Отложив в сторону рамки на мгновение даже такая нишевая задача, как бегунки задач для создания ваших JS-проектов, были кардинально изменены за последние несколько лет. Можно вспомнить переход с Grunt в 2012 году на Gulp всего пару лет спустя, и теперь есть тенденция к минимизации, используя Node.js. Менеджер пакетов NPM для запуска скриптов сборки. Говоря о менеджерах пакетов, разработчики колебались между NPM, Bower и Yarn для запуска интерфейсных зависимостей проектов. Инструменты сборки и менеджеры пакетов это небольшие, но важные части рабочего процесса веб-разработки. Однако такой же отток происходит и с тем, как на самом деле строятся приложения и пользовательский интерфейс, который, возможно, является самой центральной и важной частью веб-развития. Индивидуальному разработчику, это определенно может быть сложно уследить, хотя интересно изучить новый фреймворк или библиотеку. У некоторых «кривая» обучения более крутая, чем у других, и во многих случаях изучается «система» фреймворка, а не фундаментальные понятия HTML / JS / CSS. У разработчика в команде или в компании есть дополнительные проблемы. В начале проекта, необходимо договориться о том, какие инструменты будут использоваться для разработки в течение жизненного цикла проекта. В это входит: инструменты сборки, инструменты тестирования и, конечно же, любые фреймворки или библиотеки. Не каждый согласится с общим выбором. Если команда большая и работает над множеством проектов, может возникнуть соблазн позволить разработчикам в каждом проекте выбирать свои собственные инструменты. В конце концов, следует хорошо проанализировать потребности проекта и использовать соответствующие инструменты. В конце концов, используя разные инструменты и фреймворки могут обернуться проблемой для команды. Если все согласятся, охотно или нет, с одними и теми же рамками, все может закончиться хорошо. Но даже в этом случае, через два-три года, структура может устареть. Из-за этого устаревшие технологии начинают казаться немного удушающими, особенно для младших разработчиков в команде, которые хотят поддерживать свои навыки актуальными на ровне остальной части веб-сообщества. И в этом случае, организация стоит перед выбором: переделать весь стек технологий, используя новые решения или оставить старые, но сталкиваясь с восприятием не инновационного места работы. Это большая проблема, и ее решение обязательно! Конечно, возникает вопрос: "Какая есть альтернатива?".

Одно из самых больших преимуществ отказа от фреймворка - возможность сосредоточиться на основных концепциях веб-разработки, а не на изучении конкретных фреймворков ведь это навыки, которые могут или не могут быть перенесены в следующую популярную структуру. Еще одно огромное преимущество - возможность попробовать небольшие библиотеки и микрофреймворки, которые решают очень специфические потребности проектов. Барьер доступа к ним и даже к новым интерфейсным инструментам сборки намного ниже, учитывая, что исключается борьба с конкретной средой разработки, предоставляемой последними популярными фреймворк. Современные фреймворки чрезвычайно полезны и решают некоторые большие проблемы, но почему бы не задуматься об использовании так называемого чистого JavaScript, учитывая желание попробовать другие вещи. Взглянув на опрос State of JS, проведенный в 2017 году. Можно заметить, что разработка без фреймворка является второй по популярности после React. Однако, не совсем понятны конкретные причины, по которым разработчики отдают предпочтение не-фреймворкам, а чистому JS. Что строят разработчики? Какие инструменты и/или процессы они используют? Невозможно не восторгаться новыми технологиями и техниками, и как следствие будет нецелесообразно обойти вниманием одну из них. Веб-компоненты стали чем-то иным для веб-разработки. Особенно в такие моменты, когда создается некая часть приложения как например, с использованием 3Dweb, или просто какой-то необычный «веб-эксперимент», в котором, для простоты и универсальности удобнее использовать простой JS, CSS и HTML конкретного пользовательского интерфейса. Но раньше не было отличного способа организовать код, создавая повторно используемые компоненты, при этом не возвращаясь к фреймворку. Это именно то, как веб-компоненты и новые функции JS меняют взгляд на веб-разработку. Для начала следует рассмотреть, как новый синтаксис класса JS можно использовать при создании собственного настраиваемого элемента с использованием API веб-компонентов. А после продолжить изучать способы улучшения повторного использования, инкапсуляции и рабочего процесса: использование ShadowDOM, создание HTML шаблона, создание приложения только для компонентов и т.д. Однако, в настоящее время, веб-компоненты все еще являются чем-то новым, и веб-сообщество будет придумывать новые способы улучшения рабочего процесса. Без структуры исходный код превращается в так называемый «спагетти код». Написание новых функции могут быть огромной проблемой без предсказуемой организации проекта. Тем не менее, в попытке освободиться от больших, всеобъемлющих фреймворков веб-компоненты предоставляют огромную возможность развития в этом направлении. Прежде чем продолжать углубляться в данную тему имеет смысл выбрать в качестве наглядного примера псевдо задачу, для реализации технологии. Например, средство выбора даты в браузере. Наверняка это то с чем сталкивались, если не все, то по крайне мере большинство веб-разработчиков. Хотя сам по себе это не веб-компонент, он очень похожа концепция, если заглянуть внутрь.

Популярные современные фреймворки в основном предлагают возможность повторного использования кода в виде компонентов или модей. Вообще говоря, это отдельная часть кода (HTML / JS / CSS), которая предлагает визуальный стиль, интерактивность и, возможно, имеет API или параметры. Все эти свойства уже предоставляет браузере, ведь в нём уже есть многоразовые, модульные элементы, которые предлагают стиль, интерактивность и поставляются с API.

Конечно, речь идет о тегах HTML или элементах DOM. Они отображаются в DOM и имеют очень специфический тип функциональности. Тег <div> или тег <span> довольно общий и используется для хранения текста или смеси элементов. Элементы <button> и <input> являются более конкретными по функциональности и стилю. Когда кнопка помещается в HTML-код, она выглядит как стандартная кнопка, и когда на неё нажимают, она действует как кнопка. Подобные различия в стилях и возможностях у <input>, будь то создание средства выбора даты, слайдер или поле ввода текста.

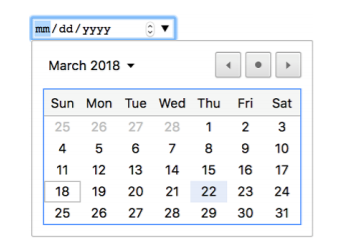


Рисунок 1.1 Расширенный пользовательский интерфейс выбора даты

Чтобы создать средство выбора даты, необходимо просто поместит следующий тег в HTML документе: <input type= "date">. То, что в действительности получается от этого простого тега, довольно сложно, но все это делает браузер сам. Этот тег (при использовании типа «дата») предлагает текстовое поле ввода, но можно нажать на месяц, день или год и перейти вверх или вниз по любому из них. Кроме того, если щелкнуть стрелку вниз сбоку, откроется представление календаря (рис. ^). Пользователь может взаимодействовать с этим интерфейсом, чтобы выбрать дату. Кроме того, в мобильном телефоне он работает немного иначе. Он не откроется таким образом, а вместо этого покажет модальное окно для выбора даты. Более того, у средства выбора даты есть свойства, которые можно запросить, включая выбранное значение. Это можно увидеть используя свойства в консоли JavaScript.console.log(document.querySelector ('ввод').значение). Средство выбора даты - отличный пример многоразового «компонента» или «модуля» с довольно сложными визуальными стилями и шаблонами взаимодействия, которые были запрограммированы в браузер разработчиками. Они работают в самых разных ситуациях. Это также отличный пример популярной веб-компонентной концепции, называемой Shadow DOM.

Shadow DOM (теневая модель документа) - часть документа, реализующая инкапсуляцию в DOM дереве. Она является частью документа и встраивается непосредственно внутрь страницы. При открытии инструментов разработчика, чтобы посмотреть на DOM, будет виден просто тег <input type=”date”>. Однако, если использовать Chrome и включить функцию отображения «[User Agent Shadow DOM](https://www.google.com/search?sxsrf=ALeKk021u3PENfO7lsbqQ0YrqlIA3PPqNQ:1612424858028&q=User+Agent+Shadow+DOM&spell=1&sa=X&ved=2ahUKEwik3Y7X3s_uAhUm4YUKHTHaAsIQkeECKAB6BAgNEDM)» в настройках инструментов разработчика, тот же тег расширяется, и выглядит так (рис. V).



Рисунок \_ Включение настроек Shadow Root в инструментах разработчика Chrome

В этом скрытом «теневом корне» гораздо больше разметки. Было бы здорово увидеть это в HTML и CSS. Eго нет, потому что этот элемент пользовательского интерфейса является частью браузера и привязан к ОС. Тем не менее, есть изрядное количество элементов, спрятанных в Shadow DOM, которые появляются в элементе поля ввода. Присмотревшись, можно заметить, что Shadow DOM содержит смесь <div> и <span>. Может прийти в голову, что это опасно. Что ж, в CSS приложения я можно очень просто определите все теги <div>, чтобы они имели синий фон с очень большим размером шрифта и все теги <span> для отображения с непрозрачностью 10%. Это бы прекрасно могло работать, за исключением одного «но»! Shadow DOM защищает внутреннюю работу веб-компонента извне. Поэтому эти синие и большие стили в тегах <div> и <span> не проникнут в Shadow DOM.

let myElement = document.getElementById ('очистить');

Более того, можно было бы написать некоторый Javascript, чтобы попытаться получить и управлять кнопкой очистки средства выбора даты (пример кода выше). При попытке получить этот элемент, поскольку он находится в пределах Shadow DOM, объект элемента не будет найден, а переменная myElement имеет значение null.

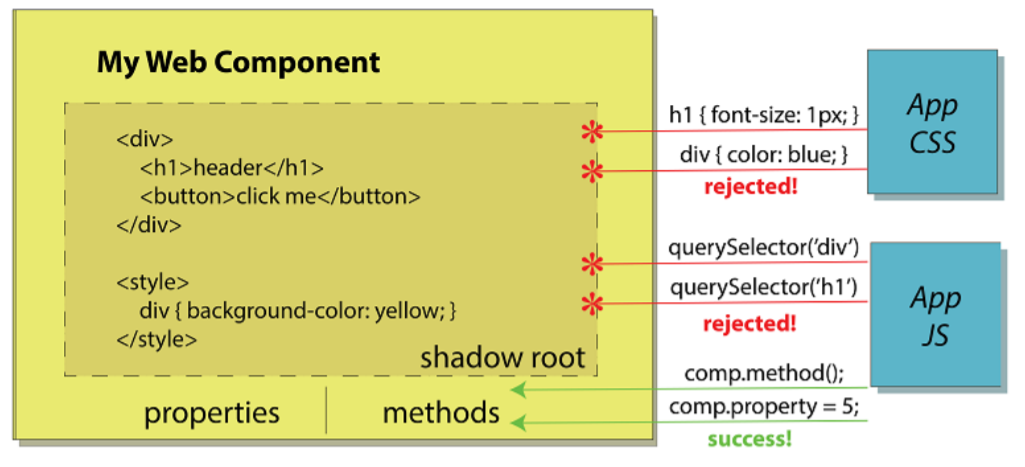


Рисунок 1.3 Shadow DOM защищает наш компонент от непредвиденных последствий, когда CSS или Javascript может повлиять на стили и узлы внутри, которые не предназначены для изменения

Это и делает Shadow DOM, защищает теневую область видимости. Можно использовать этот «теневой корень» где угодно. Это имеет смысл в настраиваемом элементе, который создается во избежании непреднамеренного сбоя, когда разработчик устанавливает правило CSS, которое имеет то же самое имя, как и то, что используется в созданном компоненте. Или же когда тот же разработчик запрашивает элемент по классу, и что-то в настраиваемом элементе случайно выбирается. Становится понятным что, выбор даты - очень полезный элемент, как и много других полезных элементов, которые используются ежедневно. Многие элементы используются для семантических целей, как например, тег <footer>. Но другие элементы имеют особый API и стиль, например, теги <button>, <option> или <video>

Каким бы красивым и удобным ни был выбор даты и любой другой элемент, было бы замечательно, если бы была возможность создавать собственные элементы с собственным визуальным стилем, внутренней логикой, возможностью повторного использования и инкапсуляция. Именно это имеется в виду, когда речь идет о веб-компонентах. В добавок инкапсуляция, обеспечиваемая Shadow DOM, может использовать API Custom Element для создания собственных компонентов, которые делают вещи, для специфичных нужд. Это возможность взять то, что интересно и создать объект многоразового использования, которым можно поделиться со всем миром, своей командой или пользоваться самому, используя его в нескольких проектах там, где это необходимо. Или даже лучше, может некий разработчик создал веб-компонент для чего-то, что нужно другому, у которого нет времени или опыта, чтобы создать это самому. Разработчики могут делиться своими компонентами, и просто использовать их его как обычный элемент DOM не зависимо от фреймворка.

Говоря о конкретных технических функциях, которые предлагают веб-компоненты, видение начало разваливаться после первоначальной шумихи вокруг веб-компонентов. Это прижилась несколько лет назад. Примерно в 2015 году было широко распространено мнение, что стандартный веб-компонент будет создан с помощью имплементирования трех новых функций:

* Пользовательские элементы;
* Теневой DOM;
* Импорт HTML.

Эта концепция никогда не принималась в качестве стандарта. Фактически, вначале Google в значительной степени отвечали за создание рабочих проектов веб-компонентов. Они взяли на себя создание API-интерфейсов и отправили их в Chrome как обнадеживающий эксперимент, чтобы увидеть, получат ли распространение веб-компоненты. HTML-импорт так и не появился.

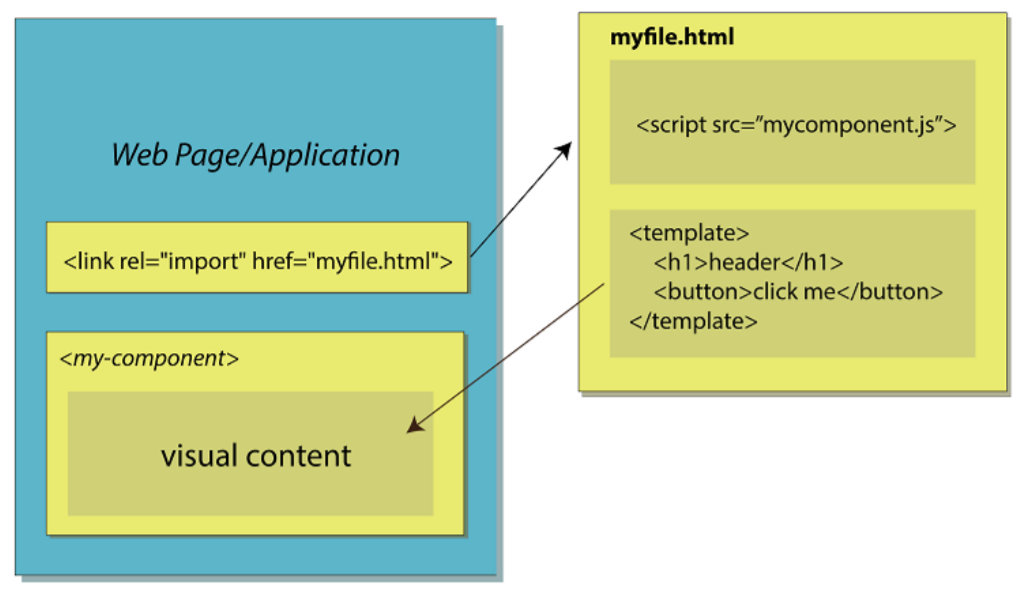


Рис. \_

С помощью импорта HTML файл, содержащий определение вашего компонента и разметку вашего компонента, может быть импортированным прямо в документ Shadow DOM (рис. ^). Другие поставщики браузеров в то время не планировали выпускать эту функцию. Firefox, в частности, хотели подождать, чтобы увидеть, насколько большой будет всплеск модулей ES6 / ES2015 и, возможно, когда-нибудь импортировать не только JS, но и HTML. HTML-импорт был довольно большой потерей. С самого начала в планах Google от него зависели веб-компоненты. HTML Import был фрагментом HTML для объявления разметки или структуры компонента, а также включал JS, определяющий логику составной части. Импорт HTML был основной точкой входа для веб-компонентов, и без них не было понятно, как вообще использовать веб-компоненты с разметкой и стилем. Единственным браузером, который принял его, был Chrome. Даже сейчас это просто Chrome и Safari. Хотя Firefox поставляет его за флагом конфигурации, а в Edge он в настоящее время находится в разработке. И Shadow DOM, и API Custom Element также перешли с версии 0 на 1. Для Custom Elements это было немного неприятно, учитывая, что разработчики, знакомые с веб-компонентам в то шаткое время надеялись перейти на новый API. Учитывая все это, разработчиков, которые назвали веб-компоненты «нарушенным обещанием» и продолжили пользоваться фреймворками вряд ли можно критиковать. В 2015 году было немного сложно правильно работать с ними, особенно при ориентации на браузеры, отличные от Chrome. Полимер и X-теги еще один аспект того, что имелось в виду, когда говорилось о веб-компонентах. Были фреймворки, появившиеся в то время, которые использовали веб-компоненты в качестве своих основных структур. Из-за нестабильности, окружающей простые компоненты без каркаса в то время, Polymer Google и X-теги Mozilla были даже более тесно связаны с веб-компонентами, чем лежащие в основе технологии. Как библиотека, Polymer довольно хороша. Она основана на главных концепциях веб-компонентов и предлагает отличные помощники и инструменты. Некоторые из них, например, lit-html и PolymerCLI могут работать даже с веб-компонентами, не основанными на полимерах. Несмотря на твердую версию 3.0 сейчас, первые дни Polymer до версии 1.0 были немного шаткими. Как и ожидалось с любой библиотекой до версии 1.0, API-интерфейсы немного изменились, особенно когда они пытались сохранить спецификации с отсутствием Shadow DOM во всех браузерах, кроме Chrome. Особенно сложно было иметь дело с Shadow DOM. Полнофункциональные полифиллы, в которые включенная инкапсуляция CSS была слишком сложной и влияла на производительность. Чтобы компенсировать это, «Shady DOM» был изобретен как облегченная реализация, которую можно было заполнить полифиллами. Это было непростое время для веб-компонентов в целом, и Polymer казался еще одним фреймворком / библиотекой, который должен был конкурировать с более устоявшимися, не имевшими дело с этими веб-стандартами.

Можно успешно использовать современные веб-компоненты для проектов, но не быть полностью удовлетворенными ими, или начать использовать некоторые совершенно новые языковые функции JS. Функция жирной стрелки оказалась отличным способом управления прицелом при работе с событиями мыши или таймером. Что еще более важно, концепция «модулей» и ключевое слово import стали удобными. Благодаря импорту можно избавиться от хрупкого беспорядка, связанного с тем, чтобы каждый используемый файл JS, был связан в теге сценария на главной HTML-странице. Каждый веб-компонент полностью отвечает за импорт собственного кода. Это означало, что на главной HTML-странице можно иметь один тег сценария на основе модуля, импортирующего веб-компонент, содержащий всё приложение. Каждый дочерний компонент просто импортирует все, что ему нужно. Это открыло двери для многоразовых модулей кода, написанных на чистом JS, и дало возможность для создания нескольких уровней наследования, когда требуется, чтобы компоненты разделяли API и были немного умнее, чем базовый HTMLElement API. Наконец, можно хранить свой HTML/CSS в отдельный файл template.js, который можно импортировать, сохраняя мои визуальные проблемы отдельно от логики контроллера компонента. Последней «огромной» функцией JS, с которой приятно работать с веб-компонентами - «Шаблонный литерал». Можно не только хранить свой HTML / CSS в отдельном файле шаблона, но и заменить выражения-заполнители в разметке на переменные, а также вложить несколько шаблонов вместе с использованием функций JS. Эти функции ES6 / ES2015 внезапно сделали работу с веб-компонентами «счастьем». Даже ранее с устаревшим HTML-импортом, можно было подумать, что комбинация модули и литералы шаблонов - лучший способ. Правда, это дает дочерним элементам компонента некоторую хорошую защиту от стиля и JS, вкрадывающихся и оказывающих неблагоприятное воздействие, но это новое решение проблемы, которая всегда присутствовала. Как бы там ни было в будущем с веб-компонентов, на данный момент не слышно ни о каких альтернативных подобных решениях. Как только Shadow DOM будет работать повсюду, это станет важным дополнением к общему набору инструментов. Никогда не бывает легко предсказать будущее, особенно в Интернете, где все меняется в безумном темпе. Тем не менее, есть несколько веских предпосылок, указывающих, куда могут двигаться веб-компоненты. Уже были видны эксперименты с React, Angular, и Vue при упаковке компонентов в каждой из этих платформ как отдельный веб-компонент. Дополнительно инструменты такие как StencilJS и Svelte позволяют создавать и компилировать в автономные веб-компоненты. Это значит, что вскоре можно будет создавать компоненты без фреймворка или с рамками по индивидуальному выбору.

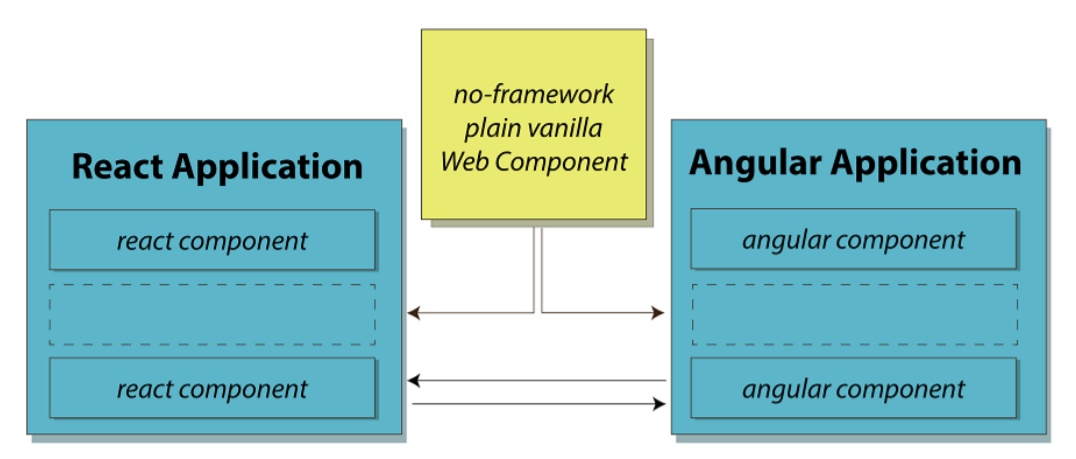


Рис. \_.

Веб-компоненты могут ликвидировать разрыв между популярными фреймворками (рис. ^). Веб-компоненты фреймворка можно использовать в этих фреймворках, но уже есть проекты, компилирующие компонент React, Angular и Vue для независимого запуска компонентов, которые можно использовать где угодно. Эта концепция может даже расшириться, позволив совершенно разным языкам работать вместе. В одном приложении могут быть разные компоненты, разработанные на Javascript, Typescript, и Coffeescript, и поскольку каждый из них представляет собой модульный компонент, предоставляющий API, это не имеет значения. Еще более безумно то, что с появлением WebAssembly стало возможным видеть такие языки, как C ++, Lua, Go и др., скомпилированными в байт-код и обернутый веб-компонентом, выглядящий как полностью нормальный элемент снаружи, одновременно обеспечивающий высокую производительность графики, которая может работать быстрее, чем обычно работает JS. Скорее всего использование модулей ES6 / ES2015 и импорта изменит образ мышления индустрии о библиотеках и фреймворках. Уже видно два похожих инструмента: lit-html и hyperHTML для расширенного управления разметкой. У обоих есть модули, которые разработчики могут импортировать вместо того, чтобы загружать всю библиотеку и решать очень конкретную проблему. В связи с этим, в будущем появятся еще много удивительных библиотек. Импортируется только то, что необходимо, и только когда это нужно. Время покажет, откажутся ли основные фреймворки от функций из-за html в отдельные импорты, которые можно использовать вне фреймворка, как это сделал Polymer, но кажется это неизбежно, особенно глядя на другие языки, в которых была функция импорта навсегда. Задолго до появления веб-компонентов были отличные способы взаимодействия с обычными элементами DOM. Мы можем использовать эти же методы, чтобы придать структуру всему, что мы строим с помощью веб-компонентов, как и обычные теги <div>, <video> или <input>. Веб-компоненты такие же, как и любой другой элемент DOM.

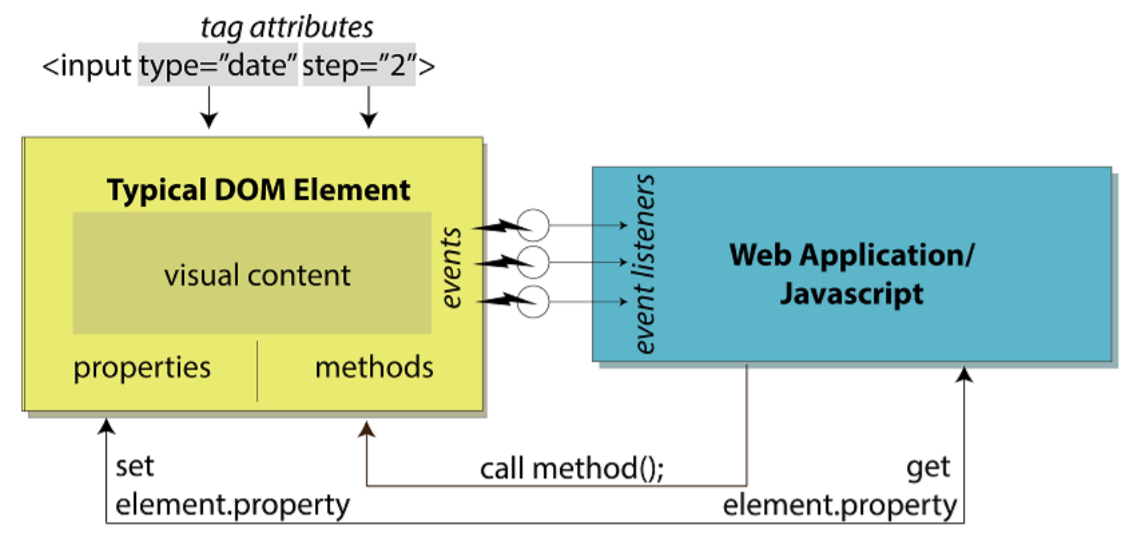


Рис. \_

Во-первых, у каждого элемента есть своего рода публичный API (рис. ^). Под этим подразумевается, возможность получить и установить свойства элемента и вызвать функции. Например, с помощью элемента видео можно вызывать функции «[pause()](https://www.w3schools.com/tags/av_met_pause.asp)» и «[play()](https://www.w3schools.com/tags/av_met_play.asp)» для управления воспроизведением видео. Также можно проверить, как долго длится видео, проверив свойство «[duration](https://www.w3schools.com/tags/av_prop_duration.asp)». Наконец, чтобы перейти к текущему моменту видео, можно установить свойство «currentTime». Очевидно, что методы и функции для объектов используются в программировании повсюду. Элементы DOM ничем не отличаются - и, кроме того, пользовательские веб-компоненты тоже. Что-то столь же простое, как тег <img>, имеет атрибут «src», который указывает элементу на расположение изображения. Веб-компоненты имеют API, так что можно внутренне отслеживать изменения атрибутов. Наконец, можно прослушать изменения из пользовательского веб-компонента. Обычно делается следующие, чтобы слушать click: mybutton.addEventListener('click', functionToCall). Также можно создавать и отправлять собственные «настраиваемые события».

Говорить об отдельных компонентах - это одно, но как насчет, создания всего веб-приложение. Веб-компоненты могут быть настолько большими или маленькими, насколько это необходимо. Можно создать несколько очень детализированных компонентов, например кнопку, а затем вложить их внутри более крупного веб-компонента, такого как собственная панель инструментов. Компонент панели инструментов может обрабатывать более тонкие детали работы с кнопками, возможно, включать и выключать их или отключать определенные при определенных обстоятельствах.

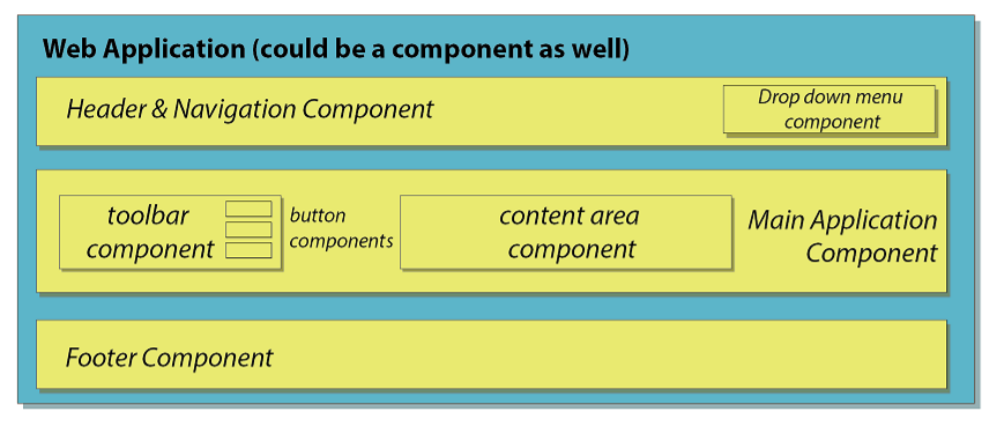


Рис. \_

Панель инструментов, наряду с другими компонентами, может быть дополнительно вложена в другой родительский элемент, компонент и так далее (рис. ^). Веб-компоненты и JavaScript без фреймворка могут многое предложить для веб-приложений. Но по мере роста приложения оно будет усложняться. Будет труднее координировать взаимодействие компонентов друг с другом. Иногда обнаруживается, что даже с внутренней структурой, которую предоставляют веб-компоненты, недостаточно для создания сложного приложения. И может возникнуть соблазн обратиться к популярным фреймворкам и библиотекам, помогающие структурировать. Популярные фреймворки, такие как Angular предлагают привязку данных, шаблоны MVC и многое другое. Безусловно, они могут быть полезны при создании традиционного веб-приложение. С другой стороны, можно писать и импортировать простой JS-код на основе проверенных и надежных шаблонах проектирования, которые существуют уже много лет, избегая этих больших рамок. Например, собственные события DOM могут не подойти. Часто есть необходимость в одной части веб-приложения для сообщения совершенно в другую часть приложения, и не хочется беспокоиться о том, как событие проходит через DOM. Можно обратиться к библиотекам на подобе RXjs или Redux, но это может оказаться излишним.

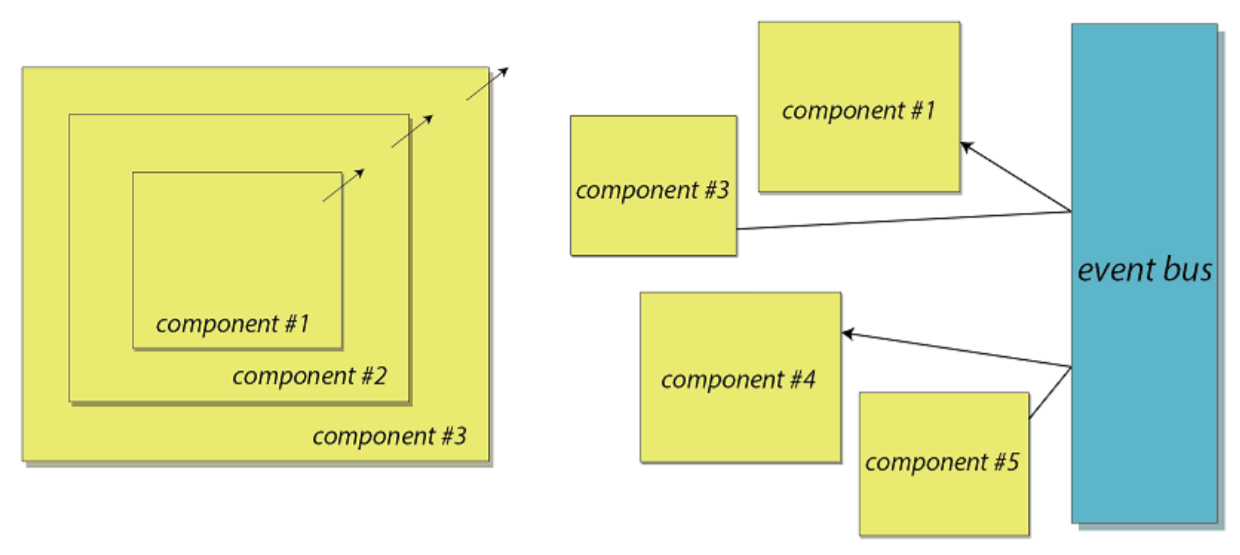


Рис. \_

Вместо этого можете написать простую шину событий с небольшим объемом кода (рис. ^). Также есть золотая середина с микрофреймворками. Микрофреймворк может быть отличным, минималистичным способом организовать приложение и добавить определенную функциональность более широкой структуре.

В конце концов, несмотря на то, что есть отличные аргументы в пользу веб-компонентов без фреймворка, проект и команда в конечном итоге повлияют на то, что используется для создания продукта. В любом развивающемся стандарте веб-компоненты пока не предлагают всех ответов.

Бывают случаи, когда веб-приложение очень простое и современный фреймворк может быть идеальным ответом, потому что он обрабатывает все, что нужно. Иногда приходится работать над типом проекта, в котором фреймворки просто мешают. В решениях, из которых можно выбрать, широкий спектр вариантов, некоторые из которых пересечение. Даже если веб-компоненты без фреймворка не подходят, фреймворк, вероятно, однажды будет построен с их помощью, даже если это не очевидно. Ознакомление с основами веб-стандартов любого фреймворка - это всегда отличная идея, даже если они не используются напрямую.

## Что такое Angular Elements

Angular Elements - это пакет в Angular, который помогает публиковать компоненты Angular как пользовательские элементы. Для этого он берет компонент Angular и компилирует его в веб-компонент. Пользовательские элементы - это функция веб-платформы, которая в настоящее время поддерживается Chrome, Edge (на основе Chromium), Firefox, Opera и Safari, а также, доступна в других браузерах через полифилы. Пользовательский элемент расширяет HTML, позволяя определить тег, содержимое которого создается и контролируется кодом JavaScript. Браузер поддерживает CustomElementRegistry? определенные настраиваемые элементы, которые сопоставляют создаваемый класс JavaScript с тегом HTML. @angular/elements пакет экспортирует createCustomElement() API. Преобразование компонента в настраиваемый элемент делает всю необходимую инфраструктуру Angular доступной для браузера. Создание настраиваемого элемента является простым и понятным, и оно автоматически связывает определяемое компонентом представление с обнаружением изменений и привязкой данных, сопоставляя функциональность Angular с соответствующими эквивалентами в собственном HTML.

**Простой динамический контент в приложении Angular -** преобразование компонента в настраиваемый элемент обеспечивает простой путь к созданию динамического HTML-содержимого в приложении Angular. HTML-контент, который добавляется непосредственно в DOM в приложении Angular, обычно отображается без обработки Angular, если не определяется динамический компонент, добавляя собственный код для подключения тега HTML к данным приложения и участвует в обнаружении изменений. С настраиваемым элементом вся эта проводка выполняется автоматически.

**Приложения с богатым содержанием -** если в приложении с богатым контентом, настраиваемые элементы позволяют предоставить поставщикам контента сложные функции Angular, не требуя знания Angular. Например, руководство по Angular, добавляется непосредственно в DOM с помощью инструментов навигации Angular, но может включать в себя специальные элементы, подобные тем, <code-snippet> которые выполняют сложные операции. Все, что нужно сообщить поставщику контента - это синтаксис настраиваемого элемента. Поставщику контента не нужно ничего знать об Angular или о структурах данных или реализации компонента.

Используется [createCustomElement](https://angular.io/api/elements/createCustomElement)() функция для преобразования компонента в класс, который можно зарегистрировать в браузере как настраиваемый элемент. После того, как регистрируется, сконфигурированный класс в реестре настраиваемых элементов браузера, можно использовать новый элемент так же, как встроенный элемент HTML в контенте, который добавляется непосредственно в DOM:

<my-popup message="Use Angular!"></my-popup>

Пользовательские элементы загружаются сами по себе - они запускаются автоматически при добавлении в DOM и автоматически уничтожаются при удалении из DOM. После добавления настраиваемого элемента в DOM для любой страницы он выглядит и ведет себя как любой другой элемент HTML и не требует каких-либо специальных знаний терминов Angular или соглашений об использовании.



Рис. \_

Когда настраиваемый элемент размещается на странице, браузер создает экземпляр зарегистрированного класса и добавляет его в DOM. Содержимое предоставляется шаблоном компонента, который использует синтаксис шаблона Angular и отображается с использованием данных компонента и DOM (рис. ^). Входные свойства в компоненте соответствуют входным атрибутам для элемента.

## Преобразование компонентов в пользовательские элементы

Angular предоставляет [createCustomElement](https://angular.io/api/elements/createCustomElement)() функцию преобразования компонента Angular вместе с его зависимостями в пользовательский элемент. Функция собирает наблюдаемые свойства компонента, а также функции Angular, необходимые браузеру для создания и уничтожения экземпляров, а также для обнаружения изменений и реагирования на них. Процесс преобразования реализует [NgElementConstructor](https://angular.io/api/elements/NgElementConstructor) интерфейс и создает класс конструктора, который настроен на создание экземпляра самозагрузки вашего компонента. Используется функция JavaScript, customElements.define(), чтобы зарегистрировать сконфигурированный конструктор и связанный с ним тег настраиваемого элемента в браузере CustomElementRegistry. Когда браузер встречает тег для зарегистрированного элемента, он использует конструктор для создания экземпляра настраиваемого элемента.

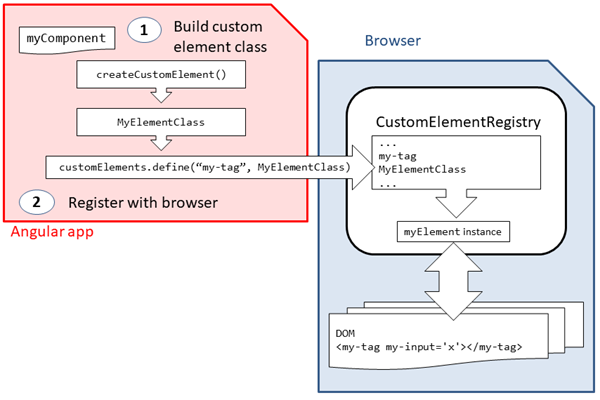


Рис. \_

Пользовательский элемент содержит компонент Angular, обеспечивая мост между данными и логикой, определенной в компоненте, и стандартными API-интерфейсами DOM. Свойства и логика компонента отображаются непосредственно в атрибуты HTML и систему событий браузера. API создания анализирует компонент в поисках входных свойств и определяет соответствующие атрибуты для настраиваемого элемента. Он преобразует имена свойств, чтобы сделать их совместимыми с настраиваемыми элементами, которые не распознают регистр символов. В именах полученных атрибутов используются строчные буквы, разделенные тире. Например, для компонента с соответствующим настраиваемым элементом определяется атрибут @[Input](https://angular.io/api/core/Input)('myInputProp') inputPropmy-input-prop. Выходные данные компонентов отправляются как [настраиваемые события](https://developer.mozilla.org/en-US/docs/Web/API/CustomEvent) HTML, при этом имя настраиваемого события совпадает с именем вывода. Например, для компонента с соответствующим настраиваемым элементом будут отправляться события с именем «valueChanged», а отправленные данные будут сохранены в свойстве события. Если указывать псевдоним, используется это значение. Например, приводит к отправке событий с именем «myClick».

@[Output](https://angular.io/api/core/Output)()valueChanged=new [EventEmitter](https://angular.io/api/core/EventEmitter)();  
@[Output](https://angular.io/api/core/Output)('myClick')clicks = new [EventEmitter](https://angular.io/api/core/EventEmitter)<string>();

Недавно разработанная функция веб-платформы [пользовательских элементов в](https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_custom_elements) настоящее время изначально поддерживается в ряде браузеров (рис. V).

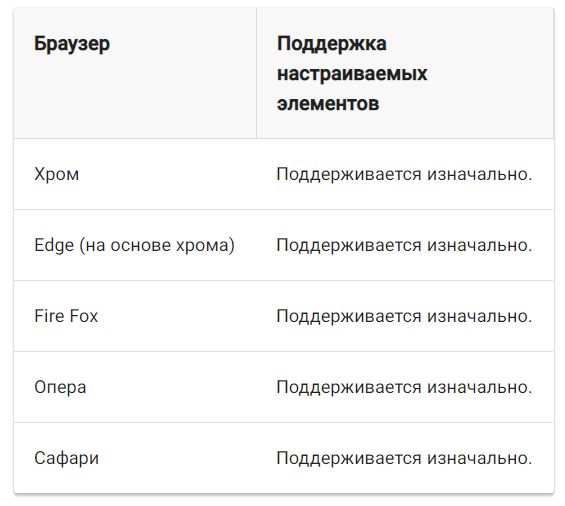


Рис. \_

В браузерах, которые изначально поддерживают пользовательские элементы, спецификация требует, чтобы разработчики использовали классы ES2015 для определения пользовательских элементов - разработчики могут отказаться от этого, установив target: "es2015" свойство в [файпле конфигурации TypeScript](https://angular.io/guide/typescript-configuration) своего проекта .



Рис. \_

Рекомендуется использовать [Angular CLI](https://angular.io/cli) для автоматической настройки проекта с правильным полифилом (рис. ^). Поскольку поддержка Custom Element и ES2015 может быть доступна не во всех браузерах, разработчики могут вместо этого использовать полифил для поддержки старых браузеров и кода ES5. Раньше, когда требовалось добавить компонент в приложение во время выполнения, нужно было определить динамический компонент, а затем загрузить его, прикрепить к элементу в DOM и подключить все зависимости, изменив обнаружение и обработку событий. Использование настраиваемого элемента Angular делает процесс намного проще и прозрачнее за счет автоматического предоставления всей инфраструктуры и фреймворка - все, что нужно сделать, это определить тип обработки событий. Все равно нужно исключить компонент из компиляции, если не предполагается использование его в своем приложении. Универсальные API - интерфейсы DOM, такие как document.createElement() или document.querySelector(), возвращают тип элемента, соответствующий указанным аргументам. Например, вызов document.createElement('a') вернет объект HTMLAnchorElement, который, как известно TypeScript, имеет свойство href. Точно так же document.createElement('div') вернет объект HTMLDivElement, который, как известно TypeScript, не имеет свойства href. При вызове с неизвестными элементами, такими как имя настраиваемого элемента ( popup-elementв нашем примере), методы будут возвращать общий тип, например HTMLElement, поскольку TypeScript не может определить правильный тип возвращаемого элемента. Пользовательские элементы, созданные с помощью Angular, расширяются с NgElement (который, в свою очередь, расширяется HTMLElement). Кроме того, эти настраиваемые элементы будут иметь свойство для каждого входа соответствующего компонента. Например, у нашего popup-elementбудет message свойство типа string. Есть несколько вариантов, если требуется получить правильные типы для пользовательских элементов. Предположим, создается my-dialog собственный элемент на основе следующего компонента:

@[Component](https://angular.io/api/core/Component)(...)   
class MyDialog {   
@[Input](https://angular.io/api/core/Input)() content: string;   
}

Самый простой способ получить точную типизацию - это привести возвращаемое значение соответствующих методов DOM к правильному типу. Для этого, можно использовать [NgElement](https://angular.io/api/elements/NgElement) и [WithProperties](https://angular.io/api/elements/WithProperties) тип (как экспортируемый из @angular/elements):

const aDialog = document.createElement('my-dialog') as [NgElement](https://angular.io/api/elements/NgElement) & [WithProperties](https://angular.io/api/elements/WithProperties)<{content: string}>; aDialog.content = 'Hello, world!'; aDialog.content = 123; // <-- ERROR: TypeScript knows this should be a string. aDialog.body = 'News'; // <-- ERROR: TypeScript knows there is no `body` property on `aDialog`.

Это хороший способ быстро получить возможности TypeScript, такие как проверка типов и поддержка автозаполнения, для вашего пользовательского элемента. Но это может стать громоздким, если вам это нужно в нескольких местах, потому что вам нужно приводить возвращаемый тип при каждом возникновении. Альтернативный способ, который требует определения типа каждого настраиваемого элемента только один раз - это расширение HTMLElementTagNameMap, которое TypeScript использует для определения типа возвращаемого элемента на основе его имени тега (для методов DOM, таких как document.createElement(), document.querySelector() и т.д.). Теперь TypeScript может определить правильный тип так же, как и для встроенных элементов:

document.createElement('div') //--> HTMLDivElement (built-in element) document.querySelector('foo') //--> Element (unknown element) document.createElement('my-dialog') //--> [NgElement](https://angular.io/api/elements/NgElement) & [WithProperties](https://angular.io/api/elements/WithProperties)<{content: string}> (custom element) document.querySelector('my-other-element') //--> [NgElement](https://angular.io/api/elements/NgElement) & [WithProperties](https://angular.io/api/elements/WithProperties)<{foo: 'bar'}> (custom element)

В итоге с помощью Angular Elements можно поместите компоненты Angular в другие библиотеки / фреймворки JavaScript, такие как React и Vue, осуществлять передачу данных из React и Vue в компонент Angular, использовать компонент Angular в приложении AngularJS. Разработчик Angular, знают, насколько разные Angular и AngularJS.

# ЗАКЛЮЧЕНИЕ

Использование фреймворка Angular позволяет стандартизировать исходный код приложения. Использование языка программирование TypeScript внутри фреймворка даёт возможность типизации, что упрощает разработку и обслуживание исходного кода. Использование Angular CLI помогает правильно создать компоненты или другие элементы фреймворка, внести их в нужные декларации. Компоненты Angular разделяют код на блоки, которыми легко манипулировать так как они не влияют на изменения друг друга. У Angular высокий порог вхождения, но последующая разработка окупает всё потраченное на обучение время. Использование декларативного программирования позволяет назначить роли объектам и обозначить поле их деятельности, поэтому у каждого объекта своё назначение и свои задачи. Директивы редактируют HTML код прямо в шаблоне это даёт возможность разделить логику отображения от логики обработки данных. Двусторонне связывание сокращает код отправки и приёма данных от HTML шаблона одной строки. Фреймворк сам следит за событиями браузера, изменениями модели и действиями пользователя на странице, чтобы сразу обновлять нужные шаблоны. При этом в коде JavaScript не требуется хранить ссылки на DOM-элементы и явно ими манипулировать. Мы просто описываем необходимый результат в терминах состояния модели, и нам не нужно использовать низкоуровневые конструкции. Модульность помогает сделать какую-то часть приложения автономной. Разбивка на модули позволяет загружать только нужные службы и эффективно выполнять автоматическое тестирование. Так же существует огромное количество готовых решений, которые позволяют решать довольно разнообразные задачи, используя уже готовые модули. Вокруг Angular большое комьюнити. В него входят как участники постоянной команды разработки, так и просто те, кто хотят внести свою лепту в развитие фреймворка с открытым исходным кодом. По Angular проходит множество конференций, о нем говорят и спорят в тематических ИТ-комьюнити. Приложения Angular просто тестировать.

Разработка веб приложений с помощью подобных технологий хоть по началу кажется сложной, в итоге оказывается довольно интересной. Причиной этому служит тот факт, что это современные технологии, а значит они должны заинтересовывать и вдохновлять. Специальнось информатика всегда подразумевает постоянный рост знаний в области и изучение новых технологических решений. Мы живем в лучшее из времен где имеем доступ ко всем мировым знаниям и большинству современных технологий. Опубликован большой список необходимой и понятной литературы, которую можно совершенно бесплатно загрузить на своё мобильное устройство или же распечатать (кому как удобно). Поэтому на сегодняшний день практически не бывает нерешенных задач.

Lazy-Loading оптимизирует работу приложения и повышает защиту. Например, некоторый функционал приложения не должен быть доступен для определенной категории пользователей и тогда загружать его вместе со всем приложение нецелесообразно. Lazy-Loading позволяет одностраничным веб приложениям (SPA) загружаться частями. С помощью этой технологии большое многофункциональное веб приложение можно загрузить частями. Для клиента это значит, что веб приложение загрузится быстрее, потому что загрузятся лишь необходимые компоненты. Так же это дает некую гарантию, что злоумышленники точно не найдут способ открыть данные так как они просто в данным момент отсутствуют физически.

Реактивное программирование помогает упрощать выражения и избегать дублирования кода. Например, у нас есть поле, которое используется в нескольких местах и любые его изменения должны быть зафиксированы везде где оно используется. Это тот случай, когда хорошим решение будет использовать принципы реактивного программирование. Вместо, того что бы нагромождать код одинаковыми алгоритмами мы просто можем подписаться на изменения объекта. Каждый раз, когда мы будет менять значения поля, другие компоненты, если они подписаны, будут узнавать о изменениях.

Одностраничные веб приложения (SPA) не требуют перезагрузки браузера. Это значит, что JavaScript код не будет прерываться и может хранить данные во время всего сеанса. Более того отсутствует потребность загружать страницу с данными целиком, а значит загружать можно только чистые данные. Меньший объем данных означает более быструю их загрузку. SPA оптимизированы для Chrome debugging, разработчики могут отслеживать сетевые действия, изучать элементы страниц и данные, которые с ними ассоциируемые.

На протяжении всей архитектуры фреймворка прослеживается идеология унификации. Тем не менее дизайн приложения целиком и полностью зависит от разработчика. Однако, дизайн приложения можно унифицировать с помощью готовых компонентов библиотек таких как ngx-bootstrap, Material Angular, Materialize и тд. Material Angular использует принципы Material Design разработанный Google. Material Design это единая концепция построения внешнего вида приложений, унифицирующая все продукты Google с целью их максимально лёгкого и интуитивного восприятия пользователями. Идее дизайнеров Google состоит в том, чтобв у приложений не должно быть острых углов, карточки переключались между собой плавно и практически незаметно. Принципы Material Design используются в мобильной операционной системе Android, поэтому знакомы каждому владельцу Android устройств. Веб приложение, написанное по этой идеалогии дизайном будет знакома большенству пользователей, а значит не будет дискомфортным и непонятным.

Angular это современный фреймворк, который стоит изучать веб разработчикам. Он надежен и удобен. Angular дает разработчику совершенно новое представление о разработке. Он сочетает в себе строгую надежность, но при этом очень гибкий и многофункциональный. SPA приложения, это то будущее куда на сегодняшний день стекается вся веб разработка, поэтому Lazy-Loading это важнейший шаг вперед. Модульная сборка приложения дает возможность разделять его части в зависимости от контекста, архитектуры проекта и поставленных задач.

Angular Universal - технология, которая позволяет разработчикам выполнять рендеринг приложений Angular на стороне сервера в различных сценариях. Данную технологию довольно просто использовать. Angular Universal имеет ряд ограничений и особенностей, которые необходимо учитывать при разработке, такие как: HTTP-запросы выполняются только по абсолютному URL; определение текущей платформы осуществляется с помощью метода isPlatformBrowser(); серверная часть приложения компилируется независимо от клиентской части. Обращение к API в Angular Universal осуществляется только по абсолютным URL. Но в случае, если приложение и API находятся на одном сервере, то клиентское приложение должно выполнять запросы по относительному URL, в то время как серверное приложение должно обращаться по абсолютному пути. Это проблема автоматически решается модулем Angular Express Engine. Для определения текущей платформы используется функция isPlatformBrowser(), которая принимает аргументом идентификатор платформы PLATFORM\_ID и возвращает true, если приложение работает уже в браузере. Серверная часть Angular Universal приложения компилируется в отдельную директорию *dist / [ название приложения ] / server*, а клиентская - в *dist / [ название приложения ] / browser*.

Angular Elements - это компоненты Angular, упакованные в виде настраиваемых элементов (также называемых веб-компонентами), веб-стандарт для определения новых элементов HTML независимо от платформы. С помощью Angular Elements можно сделать компоненты действительно многоразовыми. Это означает, что можно использовать компоненты Angular в других фреймворках и библиотеках, таких как React, Vue и Ember. С помощью Angular Elements можно добавить Angular, известный как интерфейсный фреймворк, в бэкэнд вашего проекта. Angular Elements также позволяет самостоятельно разрабатывать и публиковать части приложения. Используя Angular Element, все, что необходимо сделать, это динамически вставить тег, и Angular сам создаст для нас экземпляр компонента.

# БИБЛИОГРАФИЯ

1. Официальная документация фреймворка. [04.04.2018] <[https://angular.io/docs](https://angular.io/docs" \t "_blank)>;
2. **Адам Фримен** - Pro Angular 2017. [15.04.2018], ISBN 978-5-4461-0451-2;
3. **Яков Файн, Антон Моисеев** - Angular 2 Development With Typescript. [11.04.2018], ISBN 978-5-4461-0496-3;
4. Using OAuth 2.0 to Access Google APIs [19.04.2018] <<https://developers.google.com/identity/protocols/OAuth2>>;

**Адам Фримен** - Pro Entity Framework Core 2 for ASP.NET Core MVC. [11.04.2018], ISBN-13: 978-1484231494;

1. **Valerio De Sanctis** - ASP.NET Core 2 and Angular 5: Full-Stack Web Development with .NET Core and Angular. [11.04.2018], ISBN: 9781788293600;
2. **Орен Фарай** - Reactive Programming with Angular and Ngrx: Learn to Harness the Power of Reactive Programming with RxJS and Ngrx Extensions. [11.04.2018], ISBN13:978-1484226193;
3. Lazy loading.  <https://en.wikipedia.org/wiki/Lazy\_loading> [14.04.2019];
4. An object that doesn't contain all of the data you need but knows how to get it <<https://martinfowler.com/eaaCatalog/lazyLoad.html>> [05.03.19];
5. **Мартин Фаулер** (2003). Шаблоны корпоративной архитектуры приложений. Addison-Wesley. С. 200–214. ISBN 0-321-12742-0, [08.03.19];
6. Официальная документация фреймворка Ionic. [02.03.2019] <[https://ionicframework.com/docs](https://ionicframework.com/docs" \t "_blank)>;
7. Displaying Data <https://angular.io/guide/displaying-data> [04.04.2018];
8. Introduction to modules <https://angular.io/guide/architecture-modules> [05.04.2018];
9. Introduction to services and dependency injection <https://angular.io/guide/architecture-services> [06.04.2018];
10. Observables. Basic usage and terms. <https://angular.io/guide/observables>[06.04.2018];
11. HttpClient. <https://angular.io/guide/http> [06.04.2018];
12. Routing & Navigation. The Basics. <https://angular.io/guide/router> [07.04.2018];
13. Lazy Loading Feature Modules. <https://angular.io/guide/lazy-loading-ngmodules> [08.04.2018];
14. How to Lazy Loading in Ionic Angular. <https://blog.ionicframework.com/how-to-lazy-load-in-ionic-angular/> [08.04.2019];
15. How to Design Your Site to Make it Super-fast. <https://www.awwwards.com/how-to-design-your-site-to-make-it-super-fast.html>[11.04.2019];
16. ngx-quill  <[https://github.com/KillerCodeMonkey/ngx-quill](https://github.com/KillerCodeMonkey/ngx-quill" \t "_blank) > [05.04.2018];
17. ORM Lazy Loading Pitfalls. <http://gorodinski.com/blog/2012/06/16/orm-lazy-loading-pitfalls/> [12.04.2019];
18. Representational State Transfer . <https://ru.wikipedia.org/wiki/REST> [13.02.2019];
19. Вступление в Реактивное Программирование, которое вы пропустили. <https://habr.com/ru/post/279715/> [11.02.2018];
20. Хорошев А. Н. Введение в управление проектированием механических систем: Учебное пособие. — Белгород, 1999. — 372 с. — [ISBN 5-217-00016-3](https://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D1%83%D0%B6%D0%B5%D0%B1%D0%BD%D0%B0%D1%8F:%D0%98%D1%81%D1%82%D0%BE%D1%87%D0%BD%D0%B8%D0%BA%D0%B8_%D0%BA%D0%BD%D0%B8%D0%B3/5217000163). [Электронная версия 2011 г.](http://www.cfin.ru/management/controlling/sys_project.shtml) [15.04.2019];