WEBPACK

Веб-приложения, которыми мы пользуемся каждый день, состоят из сотен *HTML*, *CSS* и *JS* файлов. Поддержка кодовой базы таких масштабов была бы кошмаром для разработчика, но, к счастью, у нас есть «помощники», позволяющие автоматизировать рутинную работу, и тем самым облегчающие жизнь.

Внушающее количество таких инструментов заставляет задуматься о высоком: о производительности, стабильности приложений, поддерживаемости кодовой базы, и как все эти 33 удовольствия реализовать на вашем проекте. Здесь выясняется, что для каждого веб-приложения — свой подход к разработке. Исходя из этого выбираются инструменты-помощники.

Все эти инструменты разные и предназначены для решения разного рода задач. К ним относятся **упаковщики** и **сборщики проектов**, **минификаторы**, **линтеры** и даже **паттерны**. О каждой категории мы с вами поговорим в этом модуле и попробуем разобраться, как именно все эти винтики делают разработку радостнее.

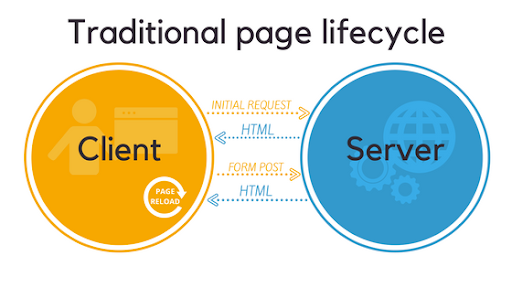
После прохождения модуля, пожалуйста, не забудьте оставить обратную связь! Это поможет нам стать лучше.

Во время изучения статей о *web*-разработке частенько можно встретить упоминание *SPA*. К сожалению, к теплым саунам эта аббревиатура не имеет никакого отношения. Настало время разобраться, что же такое — эти загадочные *SPA*, и почему последние несколько лет их популярность в разработке *web*-приложений только набирает обороты.

***SPA***— это сокращение от *Single Page Application*, то есть одностраничное приложение. Этот принцип организации *web*-приложений зародился в начале нулевых годов и с тех пор становится все популярнее.

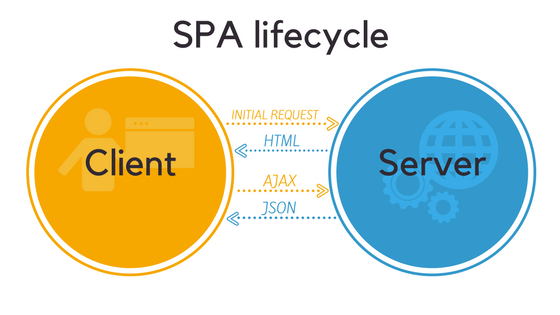
В чем особенность *SPA*? Для ответа на этот вопрос стоит разобраться, как организована работа многостраничных приложений, или *MPA*.

***MPA*** (от англ. *Multiple Page Application*), — это традиционный подход к разработке *web*-приложений, который заключается в том, что каждый раз при взаимодействии с сервером приложение получает новую *web*-страницу и отрисовывает её в браузере. Таким образом, при каждом, даже самом незначительном, изменении данных или загрузке новой информации происходит обновление всей страницы.



На картинке продемонстрировано, что после каждого запроса к серверу нам возвращается уже готовая *HTML-*страница.

Принцип *SPA*, в свою очередь, основан на идее динамической загрузки содержимого на текущую страницу без получения новой *web*-страницы с сервера. Другими словами, весь *HTML*, *JavaScript* и *CSS* могут быть получены с сервера при первой загрузке приложения, и при изменении данных будет обновляться только часть *web*-страницы, а не вся страница целиком. При таком подходе почти вся логика пользовательского интерфейса выполняется непосредственно в браузере, а взаимодействие с сервером осуществляется через *Web API*.



Здесь показано, что в отличие от *MPA*, при таком подходе сервер возвращает нам *HTML* только при первой загрузке, а в остальных случаях мы получаем от него только данные.

Важно иметь в виду, что несмотря на то, что главная страница никогда не перезагружается, приложение будет иметь различные *URL*-адреса, а также будет доступна история браузера благодаря *History API* в *HTML* 5. Одностраничные приложения считаются более инновационными и востребованными, и в дальнейшем речь пойдет именно о них, но стоит сказать пару слов об *MPA*, поскольку они имеют ряд преимуществ.

**Преимущества и недостатки**

|  |  |
| --- | --- |
| **Преимущества MPA** | **Недостатки MPA** |
| Многостраничные приложения удобны и просты для правильного управления *SEO*. Шансов на ранжирование по разным ключевым словам больше, так как *MPA* можно оптимизировать для одного ключевого слова на странице. Кроме того, на каждую страницу можно включать метатеги — это положительно влияет на рейтинг *Google*. | Время загрузки страниц увеличивается, а скорость работы снижается. |
| Первоначальная загрузка происходит быстрее за счёт того, что загружается только одна страница, а не *HTML*, *JavaScript* и *CSS* для всего приложения. | Серверная часть и клиент тесно связаны друг с другом, что осложняет процесс разработки (труднее распределить обязанности между разработчиками). |
| *MPA* обычно предоставляют множество аналитических данных и данных о том, как работает *web*-сайт. | Разработка мобильной версии приложения подразумевает написание приложения практически с нуля, поскольку, в отличие от SPA, нет возможности использовать один и тот же внутренний код. |
| Для реализации *MPA* существует обилие надёжных решений, проверенных временем, что делает этот подход экономически эффективным. | Обеспечение безопасности может стать настоящей проблемой, поскольку разработчики должны защищать каждую отдельную страницу. |
| *MPA* могут включать столько информации о продукте, сколько требуется, без ограничений по страницам. Они позволяют легко создавать новый контент и размещать его на новых страницах. |  |
| *MPA* обеспечивают понятную навигацию, что позволяет пользователю легко ориентироваться на web-сайте. |  |

Отсюда выходит, что *MPA* отлично подходят для больших информационных систем, для которых важно *SEO*, простота навигации между *web*-страницами и размещение большого количества контента.Давайте теперь попробуем разобраться, чем хороши *SPA-*приложения.

Преимущества SPA

|  |  |
| --- | --- |
| 1 | **Гибкость и отзывчивость пользовательского интерфейса**  Традиционные приложения вынуждены генерировать *web*-страницы целиком на сервере для многочисленных пользователей, что сказывается на скорости ответа сервера.  В *SPA JavaScript*, *CSS* и *HTML* загружаются только один раз во время начальной загрузки приложения, после чего сервер отправляет нам только новые данные. Это уменьшает трафик между сервером и клиентом, тем самым увеличивая скорость работы приложения. Рендеринг на стороне клиента разгружает сервер, а значит он может намного быстрее отвечать на запросы пользователей. |

|  |  |
| --- | --- |
| 2 | **Процесс разработки одностраничных приложений удобен и прост. Не нужно писать код для отображения страниц на сервере**  *SPA* подход предполагает четкие границы: сервер отвечает за данные, клиент — за отображение. Таким образом мы получаем возможность перекраивать пользовательский интерфейс по своему усмотрению, при этом никак не затрагивая сервер.  Точно так же сервер может работать с данными из разных источников различными способами, и это не повлияет на отображение.  Ну и нельзя не отметить, что подобное разделение упрощает поиск ошибок и дает возможность вести разработку клиента и сервера параллельно. |

|  |  |
| --- | --- |
| 3 | **Разработчики могут повторно использовать код**  При необходимости *SPA* можно перенести на мобильное устройство без особых трудозатрат. Дело в том, что чёткое отделение серверной части от клиента позволяет использовать один и тот же сервер не только для *web*, но также и для *Android*, и для *iOS* приложений.  Это работает и в обратную сторону: если у вас уже есть мобильное приложение, то вы можете использовать имеющиеся *API*-интерфейсы для разработки *SPA*.  Не последнюю роль также играет то, что пользовательский интерфейс *SPA* приложений близок к нативным приложениям (десктоп/мобильные). Это избавляет от необходимости полностью перерабатывать дизайн приложения: можно адаптировать существующий дизайн *SPA* приложения для мобильных устройств. |

|  |  |
| --- | --- |
| 4 | ***SPA* могут эффективно кэшировать любое локальное хранилище, а также работать в автономном режиме**  *SPA* приложения кэшируют все данные, полученные с сервера во время первоначального запроса, и хранят их локально. За счет этого при отсутствии интернет-подключения приложение сможет продолжить работу с ограниченной функциональностью. Когда подключение к сети будет восстановлено, приложение синхронизируется с сервером, обновит данные и сможет работать в полной мере. |

Недостатки *SPA*

|  |  |
| --- | --- |
| 1 | **Поисковая оптимизация**  Дело в том, что *web*-страницы индексируются через сканирование. Для сканирования поисковые системы загружают *HTML* файлы страницы, это упрощает ранжирование статических страниц. Поскольку рендеринг *web*-страницы в *SPA* выполняется на стороне клиента, они могут быть недоступны для сканирования поисковыми системами.  Так как большая часть одностраничных приложений выполняется на *JavaScript*. Это усложняет продвижение сайта. |

|  |  |
| --- | --- |
| 2 | **Первая загрузка медленная**  *SPA* запускаются медленнее традиционных приложений — это факт. Происходит это из-за этого, что в момент запуска приложение вынуждено загружать все ресурсы и скрипты. Эта загрузка может быть ощутима по времени для пользователя, особенно в больших приложениях. Однако, единовременная загрузка ресурсов необходима для рендеринга на стороне клиента.  К счастью, есть способы ускорить инициализацию *SPA*, такие как динамическая загрузка ресурсов и минимизация скриптов. |

|  |  |
| --- | --- |
| 3 | **Отключенный *JavaScript* может помешать правильной работе приложения**  Возможно, это звучит глупо, но небольшой процент людей отключает в своих браузерах *JavaScript* по разным причинам. Для таких людей *SPA* приложение будет практически бесполезным, так как может возникнуть проблема с его функциональностью. А это в свою очередь означает негативный пользовательский опыт. |

|  |  |
| --- | --- |
| 4 | **Межсайтовые сценарии могут представлять серьезную угрозу безопасности во время разработки *SPA***  Как уже было сказано, разработчики *MPA* вынуждены думать о безопасности каждой страницы. В *SPA* все, что вам нужно сделать для защиты своей страницы, — это защитить эндпоинты, что намного быстрее, но не обязательно безопаснее. Дело в том, что *SPA* более подвержены хакерским атакам, поскольку они работают на *JavaScript*, который не выполняет компиляцию кода, что делает его более уязвимым для вредоносных программ. |

Получается, что при правильной организации *SPA* не только позволяет нам предоставить пользователю отзывчивое приложение, работа с которым напоминает нативные приложения, к которым уже все привыкли, но и дает большую гибкость разработчикам. Первый вывод, который мы можем сделать: каждая архитектура имеет свои преимущества и недостатки, и выбирать *SPA* или *MPA* стоит, исходя из целей проекта.

* Если вы стремитесь предложить несколько услуг или разные категории товаров, то создание многостраничного web-приложения является логичным решением. **MPA** также являются хорошей идеей для web-сайтов с большим количеством контента (например, новостных порталов) и требующих безупречного SEO.
* Если вы вместо этого хотите сосредоточиться на продвижении одного продукта или услуги, **SPA** — лучшее решение. Они позволяют упаковать впечатляющую функциональность в легкое приложение. Кроме того, *SPA* дают возможность разрабатывать мобильные приложения на базе уже существующего кода. Таким образом данная архитектура отлично подходит для *SaaS* платформ, социальных сетей, закрытых сообществ, где поисковая оптимизация не имеет значения.

В общем, принцип *SPA* предлагает множество преимуществ, и вы можете легко преодолеть его некоторые недостатки с помощью квалифицированной команды разработчиков.

Задание 13.2.1

1/1 point (graded)

SPA — это?

Script Page Administration

Standard Pattern Account

Single Page Application верно

Отправить

В некоторых задачах доступны следующие действия: сохранение, сброс, показ подсказки или ответа. Соответствующие кнопки расположены рядом с кнопкой «Отправить».

Показать Ответ

Задание 13.2.2

1/1 point (graded)

Укажите верное утверждение:

SPA — отличный выбор для разработки web-приложений с безупречным SEO

SPA — подходит для SaaS платформ и соц.сетей, где SEO не имеет значения.. верно

Отправить

В некоторых задачах доступны следующие действия: сохранение, сброс, показ подсказки или ответа. Соответствующие кнопки расположены рядом с кнопкой «Отправить».

Показать Ответ

Последние несколько лет *SPA*-приложения постоянно расширяются, наращивают функциональность и усложняются. Очевидно, что чем большим количеством функциональных возможностей мы хотим наградить наше приложение, тем больше кода мы должны написать. Кодовая база проекта неустанно растет, увеличивается количество *JS/CSS* файлов. И тогда перед разработчиком встает вопрос:**как управлять таким количеством кода?**

На этапе небольшого приложения всё просто: мы должны в нужном порядке подключить скрипты в теле *HTML*-файла — и всё работает!

Предположим, вам необходимо загрузить файл *B* после файла *A*, а файл *C* — после загрузки обоих файлов *A* и *B*. Тогда вы, вероятно, сделаете что-то вроде этого*:*

<html>

<body>

...

<script type='text/javascript' src='path/to/A.js'></script>

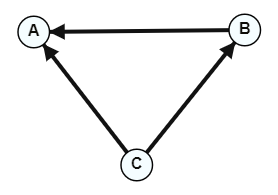
<script type='text/javascript' src='path/to/B.js'></script>

<script type='text/javascript' src='path/to/C.js'></script>

</body>

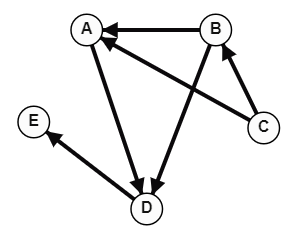
</html>

Если изобразить зависимости между файлами, мы получим такую картину:



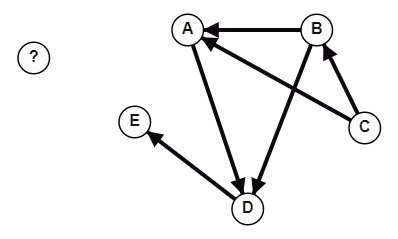
По мере роста приложения, количество таких файлов увеличивается, а зависимости становятся всё более сложными. Что, если ваш файл *B* теперь также необходимо загрузить после другого файла *D*, который, в свою очередь, необходимо загрузить после другого файла *E*, а файл *A* также необходимо загрузить после файла *D*? Вы, наверное, уже немного запутались, а у нас пока всего 5 файлов.

Для интереса попробуем изобразить зависимости между файлами на рисунке:



Мы получили дерево зависимостей — это дерево, описывающее порядок загрузки файлов.

Представим, что у нас появился новый файл, который также должен иметь своё место в порядке загрузки. Когда он должен быть загружен? Попробуйте добавить его в наш граф. Уже сложнее?



А теперь вообразите, что вы рисуете граф не для семи файлов, а для семи сотен. Сколько времени вам потребуется, чтобы разобраться с этой задачей?

Еще одна проблема заключается в том, что мы должны объединить наши файлы со скриптами в один (минимальное требующееся количество файлов) с сохранением порядка зависимостей. Это позволит избежать затрат на *HTTP*-запросы на загрузку каждого файла *JS*. Выполнение этого самостоятельно потребует много копий и вставок каждый раз, когда вы создаете или изменяете файл *JS*, одновременно рискуя ошибиться.

Для решения этих проблем нам нужен ***webpack***.

**Что такое Webpack?**

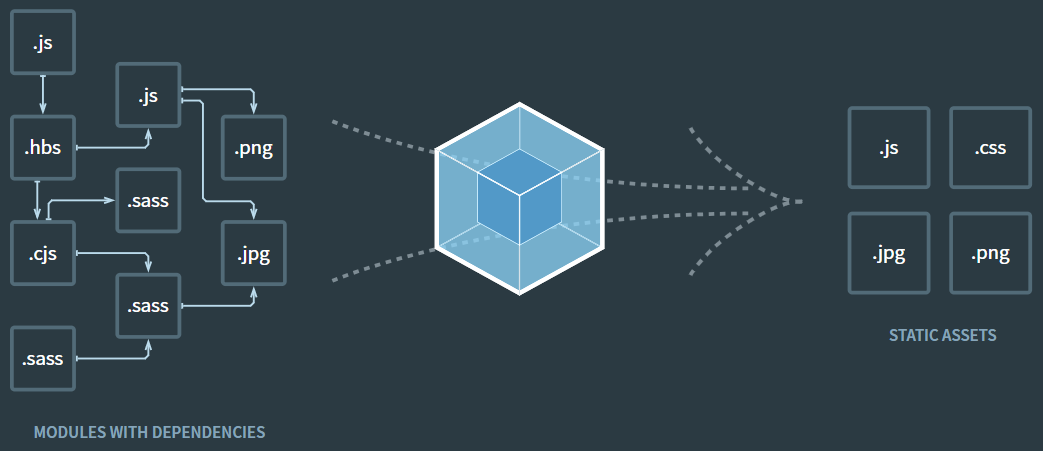
**Webpack**— это статический модульный сборщик для приложений на *JavaScript* (определение [с сайта документации](https://webpack.js.org/concepts/)).

Если говорить человеческим языком, *Webpack* соберет за вас все *JS* файлы в любое количество бандлов, соблюдая порядок их зависимостей.

**Бандл** (англ. *bundle*) — это специальный файл, в который соберет инструкции из всех*JS*-файлов (пакет со всем *js*-кодом, который был в проекте).

На данный момент *Webpack* — один из самых мощных инструментов подобного рода.

Он позволяет решать множество различных задач, таких как упаковка, компиляция и в целом организация всех ресурсов проекта. Кроме того, он имеет открытый исходный код.

Источник: [en.bem.info](https://webpack.js.org/" \t "_blank)

Но, как и любая технология, *Webpack* помимо преимуществ имеет и недостатки. Основным недостатком является то, что часть его документации устарела из-за большого количества изменений, и знакомство с работой*Webpack,* как правило, представляет некоторые трудности.

Сейчас, когда мы ответили на вопрос, что такое *Webpack,* и зачем он нужен, самое время разобраться, как же он работает.

**Как работает Webpack?**

Первое и самое важное замечание: *Webpack* работает с **модулями**. Тему модулей более подробно мы с вами раскроем в следующем разделе, а пока договоримся, что *JS*-модуль — это *JS-*файл, который явно описывает, что ему нужно в качестве входных данных и что он сделает доступным для других файлов.

Допустим наш файл *C.js* использует внешние переменные, которые описаны в файле *A.js*, и функцию, описанную в *B.js*. Значит, для корректной работы этому файлу требуется предварительная загрузка файлов *A.js* и *B.js*.

Как мы можем это обеспечить?

* 1. В первую очередь, мы избавляемся от подключений *JS-*файлов в тегах <script…> </script> в файлах *HTML*.
  2. Следующим шагом мы должны сообщить всем *JS-*файлам об их приоритете загрузки, определить порядок их загрузки. Сделать это мы можем, например, с помощью импорта в *ES6*.
  3. *Webpack*теперь собирает эти инструкции и на основе них строит ориентированный ациклический граф или, другими словами, ориентированный граф без циклов. Наличие цикла означало бы, что файл *A* зависит от файла *B*, а файл *B* зависит от файла *A*, что делает связывание невозможным.
  4. Если же циклов не обнаружено, *Webpack* берет содержимое всех этих файлов и вставляет их в один *JS-*файл в правильном порядке.

Для того, чтобы *Webpack* знал, какие же файлы нужно добавлять в *bundle*, нам необходимо определить **точку входа** в наше приложение. **Entry** (точка входа) — это *JS-*файл, который требует загрузки всех других *JS-*файлов, то есть стоит в самом конце очереди загрузки файлов. В современном сценарии это будет файл *JS*, содержащий основной/базовый компонент вашего приложения.

После того, как мы определили точку входа, остается лишь подключить этот файл в *HTML* с помощью тега <script>.

**Настройка**

Предварительно с помощью npm init мы создадим проект и добавим в него index.html и index.js. Index.js будет содержать в себе одну строку:

document.write('Hello, webpack!');

С помощью тега <script> подключим наш *JS-*файл в index.html. Проверим, что на странице выводится корректный текст.

Для работы с *Webpack* нам в первую очередь необходимо установить два пакета (*webpack* и *webpack-cli*) с помощью команды:

npm install webpack webpack-cli --save-dev

Пакет *webpack-cli* необходим нам для работы с вебпаком в командной строке.

Теперь мы можем воспользоваться командой npx webpack. Эта команда будет работать без указания точки входа и названия файла вывода, потому что у неё есть дефолтная конфигурация. В результате её выполнения вы увидите в консоли:

Built at: 08/27/2020 7:34:15 PM

Asset Size Chunks Chunk Names

main.js 970 bytes 0 [emitted] main

Entrypoint main = main.js

[0] ./src/index.js 44 bytes {0} [built]

В самом проекте при этом появится новая директория dist, в которой будет находиться main.js файл, сгенерированный вебпаком. **Именно в этот файл будет собираться весь *JS* код проекта**.

Подключим этот файл в index.html вместо index.js и обновим страницу: никаких ошибок не возникло.

Давайте попробуем изменить index.js файл: поменяем текст с

'Hello, webpack!'

на

'Hello, world!'

Для того, чтобы main.js подхватил эти изменения, нужно пересобрать проект, то есть снова запустить npx webpack в командной строке. После этого мы обновляем страницу в браузере и видим, что текст изменился. Юху, мы правильно подключили вебпак!

Чтобы каждый раз не пересобирать проект вручную, можно воспользоваться командой npx webpack --watch, которая не будет завершать процесс после сборки проекта. После того, как вы измените код, вам нужно будет всего лишь обновить страницу в браузере, чтобы увидеть эти изменения.

Также, можно создать скрипт в пакете package.json. Давайте добавим в объект scripts следующую строку:

"build" : "webpack"

Теперь мы можем запускать вебпак с помощью скрипта npm run build.

Настало время для того, чтобы создать конфигурационный файл для вебпака. Для этого в корне проекта создадим файл с названием webpack.config.js.

Конфигурации вебпака должны описываться в module.exports :

module.exports = {

entry: './src/index.js',

output: {

filename: 'main.js'

}

}

Главное, что мы должны указать здесь — это *entry*, точка входа в приложение. В данном случае это наш index.js . Объект output описывает опции для файла, который генерирует *webpack* для сборки наших модулей. В данном случае мы задаем название для файла — main.js (оно же используется по умолчанию). Предлагаю для интереса попробовать изменить название файла вывода и пересобрать проект. В результате у вас должен появится новый файл с названием, которое вы укажете в поле filename. Для дальнейшей работы вам останется только подключить его вместо main.js в *HTML-*файл.

<https://lms.skillfactory.ru/courses/course-v1:Skillfactory+FR+2020/courseware/4277fd5e8f184b7ebf2eb5b830e09fdd/900019ad62c94284969a6672e8010e4e/?child=first>

По мере того как разработка *JavaScript* становится все более распространенной, с пространствами имен и зависимостями становится все труднее работать. Для решения этой проблемы были разработаны различные решения в виде **модульных систем**.

**Модуль** — это функциональный блок, предоставляющий открытый *API*, что позволяет использовать его в любом месте приложения.

Хорошо написанный модуль имеет самодостаточный набор функциональных возможностей и обеспечивает высокий уровень инкапсуляции, что позволяет без проблем добавлять и убирать его, не нарушая при этом работу всей системы.

**Таким образом можно сформулировать основные преимущества модулей:**

* 1. **Приватные пространства имен**

По умолчанию все переменные в *JavaScript*, которые были определены не в рамках функции, считаются глобальными, а значит, являются доступными в любом месте. Работа с глобальными переменными в несвязанных между собой функциональных блоках может привести к неожиданным проблемам. Переменные, определенные в модуле, будут иметь “модульную” область видимости, то есть будут доступны только в рамках данного модуля, пока вы сами явно не сделаете её доступной для других файлов. Таким образом модули помогают избежать проблем глобального пространства имен.

* 1. **Переиспользование**

Модуль можно использовать не только в рамках одного проекта, а в любое время, когда вам будет это нужно.

Часто у продуктовых компаний не один продукт, а сразу несколько. Все они активно развиваются и поддерживаются. Допустим, вы работали на проекте А, но однажды к вам пришел тимлид и попросил помочь товарищам с другого проекта Б. Это стандартная практика, когда на проекте не хватает рук. Вам дают задание разработать функциональность, которая уже существует в вашем продукте. В таком случае, если нужная вам часть кодовой базы была “упакована” в модуль, то вы можете с легкостью подключить её к проекту Б, тем самым избавляя себя от необходимости переписывать код.

По такому же принципу работают библиотеки: в сети существует множество решений для самых разных задач, можно выбрать нужную вам библиотеку и подключить ее в ваш проект.

* 1. **Поддержка**

Любую кодовую базу можно логически поделить на функциональные области, относительно независимые друг от друга. За счет того, что модули четко разграничивают эти области и инкапсулируют их в себе, мы избавляемся от лишних зависимостей, а это значит, что изменение одного модуля не влечет обязательное изменение другого. Это упрощает разработку приложения.

**JS модули**

Существует несколько способов создать модуль в *JS*:

* + Использование замыкания
  + *CommonJS*
  + *AMD* модули
  + *ES6* модули

Самые новые — *ES6* модули. Преимущественно о них и пойдет речь, так как вам придется тесно с ними работать. Но несмотря на то, что остальные способы уже считаются устаревшими, мы познакомимся и с ними.

**Паттерн “Модуль”**

До *ES6* в *JavaScript* не было поддержки модулей, однако можно было симулировать модульную архитектуру с помощью *IIFE* (*Immediately Invoked Function Expression*).

// Immediately Invoked Function Expression

(**function**() {

console.log('This is a module!')

})()

// Пишет This is a module! в консоли и возвращает undefined

Также, можно записать результат выполнения *IIFE* в переменную:

// Объявления модуля как глобальной переменной

**var** es5Module = **function**() {

// Внутренняя логика модуля

**function** sayHi() {

console.log('Hi, user');

}

// Внешний API

**return** {

sayHi: sayHi

};

}()

В результате выполнения кода, в *es5Module* будет записан объект {sayHi: sayHi}, а значит, мы можем обратиться к методу  sayHi() через переменную.

// Доступ к функциональности модуля

es5Module.sayHi(); // 'Hi, user'

Оба подхода позволяют создать приватное пространство имен, за счет замыкания, но не позволяют управлять зависимостями.

**CommonJS**

Этот подход применяется в *Node.js*. Как и в замыканиях, при использовании *CommonJS* в *JS-*файле, модуль хранится в уникальном контексте. Мы используем специальный объект *module* и размещаем ссылку на нашу функцию в module.exports. За счёт этого *CommonJS* знает, что мы хотим открыть модуль так, чтобы другие файлы могли его использовать. После этого мы можем его использовать в других местах нашего приложения. Для этого надо подключить модуль с помощью конструкции require.

**var** dep1 = require('./dep1');

**var** dep2 = require('./dep2');

module.exports = **function**(){

// ...

}

Здесь уже, в отличие от использования замыкания, есть возможность управлять зависимостями. При работе с *CommonJS* модули загружаются синхронно.

**Асинхронное определение модуля AMD (англ. Asynchronous Module Definition).**

Что, если нам необходимо загружать модули асинхронно? Здесь на выручку приходит *AMD*. Загрузка модулей с помощью *AMD* выглядит следующим образом:

// Вызов функции define с массивом зависимостей и фабричной функцией

define(['dep1', 'dep2'], **function** (dep1, dep2) {

// Определение модуля с помощью возвращаемого значения

**return** **function** () {};

});

Define, функция определения модуля, первым параметром принимает массив зависимостей, а вторым — функцию обратного вызова, которая и описывает сам модуль.

**ES6 модули**

Наконец, в *ES6(ES2015)* был предложен стандарт собственных модулей. Внутри *ES6* модуля все по умолчанию считается приватным и работает в строгом режиме. Для того, чтобы открыть переменные, функции и классы для других модулей, необходимо использовать export.

// math.js

**const** PI = **3.14**;

**export** **const** sum = (a, b) => a + b;

**export** **const** multiply = (a) => a\*PI;

Можно использовать оператор export один раз, перечислив все сущности, которые вы хотели бы сделать публичными:

**export** { multiply, sum };

Для того, чтобы теперь использовать эти переменные и функции в своем коде, необходимо импортировать их с помощью ключевого слова import, указав относительный или абсолютный путь к файлу.

// index.js

**import** { sum, multiply } from './math.js';

console.log( sum(**1**,**2**) );

А как быть, если мы пытаемся импортировать функцию, название которой уже использовано в нашем коде? Для разрешения конфликтов имен можно использовать псевдонимы в импорте.

**import** { multiply as myFn } from './math.js';

console.log( myFn(**4**) );

Ну и наконец, можно предоставить пространство имен, импортируя все публичные элементы модуля:

**import** \* as math from './math.js';

console.log( math.multiply(**5**) );

console.log( math.sum(**1**,**2**) );

Выберите способы реализации модульной архитектуры в JS:

AMD

UMD

SPA

GeneralJS

ES6 Module

LIFO

IIFE

CommonJS

верно

Отправить

В некоторых задачах доступны следующие действия: сохранение, сброс, показ подсказки или ответа. Соответствующие кнопки расположены рядом с кнопкой «Отправить».

Показать Ответ

**CSS модули**

**CSS-модуль**— это файл *CSS*, в котором все имена классов и анимации имеют локальную область видимости. Давайте немного разберемся.

По умолчанию имена классов *CSS* и имена анимации имеют глобальную область видимости. Это может привести к конфликтам стилей, особенно в больших *CSS*. Попросту говоря, один стиль может быть перезаписан другим стилем. Именно эту проблему решают *CSS-*модули. *CSS-*классы доступны только внутри компонента, в котором он используется.

В модулях *CSS* каждый файл компилируется отдельно, поэтому вы можете использовать простые селекторы классов с общими именами — нам не нужно беспокоиться о загрязнении глобальной области.

Рассмотрим такой пример: мы разрабатываем компонент кнопки, который имеет 3 состояния: *normal*, *error*и *disabled*.

Как бы вы описали стили для этого компонента? Наверное, получилось бы что-то такое:

/\* components/submit-button.css \*/

**.Button** { /\* all styles for Normal \*/ }

**.Button--disabled** { /\* overrides for Disabled \*/ }

**.Button--error** { /\* overrides for Error \*/ }

Однако, наш файл уже называется *submit-button*, и мы не хотели бы использовать префикс с названием компонента в начале класса. Модули помогают нам в этом.

/\* components/submit-button.css \*/

**.normal** { /\* all styles for Normal \*/ }

**.disabled** { /\* all styles for Disabled \*/ }

**.error** { /\* all styles for Error \*/ }

Это становится возможным благодаря компиляции *CSS*-модулей и использования import (или required для загрузки файла из *JS*).

/\* components/submit-button.js \*/

**import** styles from './submit-button.css';

buttonElem.outerHTML = `<button **class**=${styles.normal}>Submit</button>`

Фактические имена классов генерируются автоматически и гарантированно уникальны. Модули *CSS* делают это за вас и компилируют файлы в формат, называемый *ICSS —* именно так *CSS* и *JS* могут взаимодействовать. Итак, когда вы запустите приложение, вы увидите что-то вроде:

<button **class**="components\_submit\_button\_\_normal\_\_abc5436">

Processing...

</button>

В модулях *CSS* каждый класс должен иметь все стили, необходимые для каждого отдельного состояния. Связано это с тем, как эти стили используются в *JS*.

/\* BEM Style \*/

innerHTML = `<button **class**="Button Button--in-progress">`

/\* CSS Modules \*/

innerHTML = `<button **class**="${styles.inProgress}">`

Обратим внимание на то, что в *BEM* стиле к компоненту применяются одновременно два класса, а при использовании модулей — один. Отсюда получается, что каждый класс, который описывает конкретное состояние компонента, должен дублировать все общие свойства. Но для того, чтобы нам не приходилось непосредственно дублировать одни и те же стили несколько раз, существует ключевое слово *composes*, которое говорит, что класс включает в себя все стили, описанные в указанном классе.

**.common** {

/\* all the common styles you want \*/

}

**.normal** {

composes: common;

/\* anything that only applies to Normal \*/

}

**.disabled** {

composes: common;

/\* anything that only applies to Disabled \*/

}

**.error** {

composes: common;

/\* anything that only applies to Error \*/

}

Кроме того, *CSS-*модули позволяют нам с помощью composes ссылаться на стили из другого файла:

/\* colors.css \*/

**.primary** {

**color**: **#720**;

}

/\* submit-button.css \*/

**.common** { /\* font-sizes, padding, border-radius \*/ }

**.normal** {

composes: common;

composes: primary from "../shared/colors.css";

}

Этот прием называется **композицией**.

## ****Дополнительно****

1. Более подробно про CSS-модули можно почитать в этой [статье](https://glenmaddern.com/articles/css-modules)

https://glenmaddern.com/articles/css-modules

1. [Методология CSS-модулей](http://prgssr.ru/development/metodologiya-css-modulej.html)

<https://prgssr.ru/development/metodologiya-css-modulej.html>

## ****Подключаем CSS к Webpack****

Давайте попробуем подключить наш CSS в Webpack. По умолчанию Webpack может работать только с JSON и JS-файлами.

Загрузчики позволяют вебпаку обрабатывать другие типы файлов и преобразовывать их в модули, чтобы использовать их в приложении и добавлять в граф зависимостей.

В первую очередь нам понадобятся загрузчики: style-loader и css-loader. Установим их локально с помощью:

npm i style-loader css-loader --save-dev.

Теперь нам необходимо дополнить наш конфигурационный файл webpack.config.js. В module.exports появляется еще одно свойство — module. Это объект со свойством rules, которое описывает правила для обработки файлов, типы которых по умолчанию не обрабатываются вебпаком.

module.exports = {

output: {

filename: 'my-first-webpack.bundle.js'

},

module: {

rules: [

{ test: /\.css$/,

use: ['style-loader', 'css-loader'] }

]

}

};

В use мы перечисляем загрузчики, которые будут отвечать за преобразование файлов в корректные модули. При этом загрузчики будут вызваны по цепочке в обратном порядке: то есть в данном примере сначала файлы будут обработаны с помощью css-loader, после чего результат этого преобразования попадет в style-loader.

Нужно отметить важный момент: webpack будет ждать, что последний загрузчик вернет ему JavaScript-модуль. Поэтому нужно тщательно следить за порядком загрузчиков в массиве: если в примере мы поменяем загрузчики местами, то вебпак сообщит нам об ошибке.

Чтобы вебпак понял, какие именно файлы нужно пропускать через эти загрузчики, мы определяем свойство test. Это свойство содержит регулярное выражение, которое пропускает только файлы определенного расширения. test: /\.css$/, говорит вебпаку, что этому правилу должны подчиняться только файлы, заканчивающиеся на .css.

После того, как мы дописали конфигурацию, осталось только создать файл со стилями и добавить зависимость на него в какой-нибудь JS-файл. Это особенность вебпака, которая позволяет построить более точный граф зависимостей.

Запускаем сборку проекта и наслаждаемся результатом. Но недолго, потому что если внимательно посмотреть в консоль разработчика в браузере, можно заметить, что стили были добавлены инлайном с помощью тега <style>. Это неудобно, и в принципе инлайновые стили разработчики стараются избегать. Для этого существует специальный плагин mini-css-extract-plugin. Официальная документация советует использовать его в сочетании с уже знакомым нам css-loader.

Как всегда, сначала мы устанавливаем зависимости:

npm install mini-css-extract-plugin --save-dev

Теперь нужно снова поправить webpack.config.js.

Для начала нужно импортировать всё то, что дает нам для использования установленный пакет:

**const** MiniCssExtractPlugin = require('mini-css-extract-plugin')

Затем мы меняем в массиве use style-loader на MiniCssExtractPlugin.loader . Однако, по умолчанию этот загрузчик возвращает модули типа CommonJS. Так как мы работаем с ES6 модулями, то нас этот расклад не устраивает, поэтому мы будем использовать более развернутую версию описания загрузчика.

use: [

{

loader: MiniCssExtractPlugin.loader,

options: {

esModule: **true**,

},

},

'css-loader',

],

Именно esModule: true позволяет MiniCssExtractPlugin.loader работать в ES6 модулями.

Затем нужно добавить еще одно свойство в module.exports: массив plugins. Здесь мы перечисляем плагины, которые хотим подключить к вебпаку.

plugins: [**new** MiniCssExtractPlugin()].

После сборки проекта с таким конфигурационным файлом, будет обновлен main.js и создан main.css. Естественно, можно изменить имя выходного файла и для CSS, но сейчас мы не будем заострять на этом внимание. Осталось подключить полученный main.css в HTML-файл с помощью:

<link rel="stylesheet" type="text/css" href=" main.css">

Таким образом мы настроили вебпак на сборку не только JS кода, но и CSS.

## 13.5 Минификаторы

## https://lms.skillfactory.ru/courses/course-v1:Skillfactory+FR+2020/courseware/4277fd5e8f184b7ebf2eb5b830e09fdd/900019ad62c94284969a6672e8010e4e/?child=first

В каждом приложении самое главное — **пользователь**.

Каждое из когда-либо существовавших приложений было сделано для конкретной группы пользователей, чтобы сделать их жизнь лучше: облегчить процесс выбора бытовой техники, избавить от необходимости стоять в очередях Сбербанка для оплаты коммунальных счетов, отслеживать посылки и делать множество других вещей.

Мы уже говорили о том, что чем богаче функциональные возможности приложения, тем больше его кодовая база приложения, что увеличивает время загрузки. Особенно критично это может быть для *SPA*, так как первоначальный запуск приложения предполагает полную загрузку ресурсов c сервера на клиент. Если ваше приложение будет загружаться очень долго или медленно работать, пользователь уйдет, что никак не пойдет на пользу вашему проекту.

Кодовая база любого приложения состоит из большого количества файлов, написанных разработчиками или сгенерированных инструментами. Все эти файлы имеют вес. Хороший разработчик заботится о том, чтобы его код был легко читаем и понятен. Для этого используются различные стили форматирования кода. Для человека, в отличие от компьютера, форматирование кода играет важную роль.

Так как компьютер одинаково быстро читает как отформатированный, так и неотформатированный код, мы можем улучшить размер наших файлов с помощью **минификации** на этапе сборки проекта.

**Минификация** — процесс облегчения кода за счет удаления ненужных пробелов и символов.

Дело в том, что отступы и символы, которые мы используем для форматирования нашего кода, фактически не влияют на его работу, однако увеличивают размер файла.

Что именно будет удалено из кода после минификации:

* Переносы строк
* Пробелы
* Разделители блоков
* Комментарии

Таким образом, используя минификацию, мы можем сократить время загрузки страницы, потому что файлы с кодом становятся легче. Эта процедура позволяет уменьшить размер исходного файла на 10 – 20%. Когда пользователь запрашивает веб-страницу, вместо полной версии отправляется уменьшенная версия, что приводит к более быстрому времени отклика и снижению затрат на пропускную способность.

**Пропускная способность** — скорость, с которой данные могут быть переданы на ваш компьютер с веб-сайта или интернет-службы в течение определенного времени.

Минифицированный файл будет иметь расширение *.MIN*. Например, table.min.css.

**ВАЖНО:**Минификация и сжатие — это не одно и то же. Сжатие — это процесс, в результате которого размер оригинального файла уменьшается за счет использования алгоритмов сжатия. В отличие от сжатых файлов результат минификации не нужно распаковывать для взаимодействия с ним (чтение, изменение, выполнение).

Минификация работает путем анализа и перезаписи текстовых частей веб-сайта, чтобы уменьшить общий размер файла. Минификация распространяется на сценарии, таблицы стилей и другие компоненты, которые веб-браузер использует для отображения сайта.

Если вы не используете точки с запятой, то при попытке минимизировать свой код вы получите всевозможные проблемы.

[Здесь](https://jscompress.com/) вы можете посмотреть, как будет выглядеть ваш минифицированный код.

**ИТОГ:**

* 1. Вы пишете код с соблюдением форматирования для собственного удобства и комфорта коллег, используете осмысленные названия переменных, пишете комментарии.
  2. Минификатор анализирует код и перезаписывает его, при этом удаляя всё, что не требуется для его выполнения, а также сокращая имена локальных переменных.
  3. Web-сервер использует вместо оригинального файла минифицированный, что приводит к снижению требований к пропускной способности приложения без ущерба функциональности.

**Методы и инструменты минификации**

По умолчанию в *production mode* вебпак минифицирует код. Для этого используется *TerserPlugin*. Именно поэтому в main.js отсутствуют отступы. Однако, можно вместо него использовать другие: *BabelMinifyWebpackPlugin* и *ClosureWebpackPlugin*. Убедитесь, что плагин, который вы собираетесь подключить, удаляет ненужный код, и укажите его как optimization.minimizer.

*CSS* код тоже стоит минифицировать. Для этого можно использовать плагинoptimize-css-webpack-plugin. Его также надо будет указать в optimization.minimizer.

**ВАЖНО:**Настройка optimization.minimizer отменяет все значения по умолчанию, поэтому если вы определяете плагин для минификации *CSS*, не забудьте указать, какой плагин будет отвечать на минификацию *JS*.

Таким образом в module.exports появится еще одно свойство:

optimization: {

minimizer: [**new** TerserJSPlugin({}), **new** OptimizeCSSAssetsPlugin({})],

},

**ВАЖНО:**Для webpack v5 и выше используйте [css-minimizer-webpack-plugin](https://github.com/webpack-contrib/css-minimizer-webpack-plugin" \t "_blank).

<https://github.com/webpack-contrib/css-minimizer-webpack-plugin>

## 13.6 Шаблонизаторы

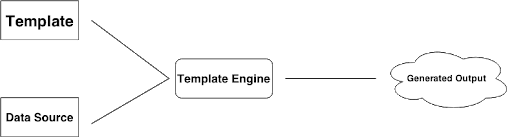
https://lms.skillfactory.ru/courses/course-v1:Skillfactory+FR+2020/courseware/4277fd5e8f184b7ebf2eb5b830e09fdd/900019ad62c94284969a6672e8010e4e/?child=first

**Шаблонизаторы** (англ. *template engines*) — инструменты для разделения модели (*model*) и представления (*view*) программы на две независимые части.

Это упрощает разработку как модели, так и представления, повышает гибкость и упрощает модификацию и обслуживание.

Шаблонизаторы позволяют управлять *HTML*-шаблонами, в которые добавляется динамическое содержимое с помощью*JavaScript*, и которые призваны генерировать конечные *HTML*-страницы.

Очень важно поддерживать строгую независимость представления (*view*) от модели (*model*), чтобы не потерять все преимущества использования шаблонизатора.



В данный момент существует большое количество шаблонизаторов для разных языков программирования. Вам остается выбрать свой. Давайте рассмотрим эти инструменты на примере *Pug*.

**Pug** — это высокопроизводительный шаблонизатор, реализованный с помощью*JavaScript* для Node.js и браузеров.

В первую очередь для работы с *Pug* нам потребуется установить *pug* и загрузчик для него:

npm install pug pug-loader --save-dev

Помимо этого нам понадобится плагин HtmlWebpackPlugin, который упрощает создание файлов *HTML* для обслуживания веб-пакетов.

Затем мы меняем webpack.config.js:

module.exports = {

entry: '/index.js',

output: {

filename: 'main.js'

},

plugins: [

**new** HtmlWebpackPlugin({template: '/index.pug'})

],

module: {

rules: [

{

test: /\.pug$/,

loader: 'pug-loader',

options: {

pretty: **true**

}

}

]

}

};

— “template” (HtmlWebpackPlugin) — путь до шаблона.

— “test: /\.pug$/” — регулярное выражение, которое говорит о том, что работать будем только с файлами, которые заканчиваются на “.pug”.

— “pretty: true” — просим pug-loader расставить отступы и переносы строк (иначе получим весь html-код в одну строку).

Файлы с расширением pug заменят нам html. Внутри это выглядит примерно так:

doctype html

html ( lang = "en" )

head

title = pageTitle

script ( type = 'text / javascript' ) .

if (foo) bar ( 1 + 5 )

body

h1 Pug - шаблонизатор узлов #container .col if youAreUsingPug

p Вы просто великолепны else

p Давай !

п.

Pug - это краткий и простой язык шаблонов с

упором на производительность и мощные функции.

В результате компиляции этот код будет иметь следующий вид:

<! DOCTYPE html >

< html lang =" en " >

< head >

< title > Мопс </ title >

< script type =" text / javascript " >

**if** ( foo ) bar ( **1** + **5** )

</ script >

</ head >

< body >

< h1 > Pug - шаблонизатор узлов </ h1 >

<div id =" container " class =" col " >

< p > Вы просто великолепны </ p >

< p > Pug - лаконичный и простой язык шаблонов с упором на производительность и мощные функции. </ p >

</ div >

</ body >

</ html >

Метод pug.compile() скомпилирует *PUG*файлы в*JS*.

//js файл

**const** pug = require('pug');

// Compile the source code

**const** compiledFunction = pug.compileFile('template.pug');

// Render a set of data

console.log(compiledFunction({

name: 'Timothy'

}));

// "<p>Timothy's Pug source code!</p>"

// Render another set of data

console.log(compiledFunction({

name: 'Forbes'

}));

// "<p>Forbes's Pug source code!</p>"

А метод pug.render() объединяет компиляцию и рендеринг в один этап. Однако, стоит помнить, что функция шаблона будет повторно компилироваться каждый раз при render вызове, что может повлиять на производительность.

const pug = require('pug');

// Compile template.pug, and render a set of data

console.log(pug.renderFile('template.pug', {

name: 'Timothy'

}));

// "<p>Timothy's Pug source code!</p>"

[Здесь](https://html2jade.org/) вы можете посмотреть, как будет выглядеть ваш *HTML* в формате *PUG*.

Предлагаем попробовать подключить шаблонизатор в проект:

Когда мы только знакомились с *JavaScript*, мы говорили о том, что это интерпретируемый язык высокого уровня с динамической и слабой типизацией. Вы, ведь, еще помните, что такое типизация, и какая она бывает? Чтобы освежить в памяти эти понятия, вернитесь в модуль 5.2 (раздел *Программирование на JavaScript: основы*)*.*

Однако последние несколько лет всё больше разработчиков прибегают к инструментам, которые добавляют статическую типизацию в *JavaScript*. Такими инструментами являются, например, *TypeScript* от*Microsoft*, появившийся в 2011 году, и *Flow* — решение от *Фейсбук\**, увидевшее свет на три года позже.

**TypeScript** — язык программирования с открытым кодом, основанный на *JavaScript* путем добавления статической типизации.

**Flow** — это библиотека с открытым кодом для статической проверки типов для *JavaScript* кода.

Прежде чем мы более подробно поговорим про эти инструменты, давайте подумаем, зачем вообще нам стоит добавлять статическую типизацию, если язык этого не предполагает по умолчанию?

Дело в том, что динамическая типизация приводит к тому, что ошибки в коде, связанные с конфликтами типов, выявляются уже в момент работы программы, и их сложно отлаживать в *IDE*. Это может очень сильно усложнить разработку.

Однако надо понимать, что, как и динамическая типизация, статическая имеет свои плюсы и минусы.

**Преимущества и недостатки статической типизации**

|  |  |
| --- | --- |
| **Преимущества** | **Недостатки** |
| Статическая проверка типов позволяет вычислить ошибки до выполнения программы. Таким образом мы можем гарантировать, что переменные и функции будут принимать значения только правильного типа. | Статические типы повышают порог вхождения в язык программирования.*JavaScript* считается один из самых подходящих языков для погружения в программирование, потому что он не требует от студента предварительного изучения системы типов для того, чтобы начать писать на *JavaScript*. Для того, чтобы эффективно использовать статическую типизацию, необходимо практиковаться. |
| Код становится более прозрачным: каждому разработчику становится очевидно, какой тип значения имеет переменная или возвращает функция. Это облегчает разработку. | Указание типов увеличивает объём кодовой базы. С одной стороны — лишние строки кода не всегда уместны, и типизация может сделать код излишне подробным, а с другой — стоит помнить, что это также является и преимуществом статической типизации. |
| Динамическая типизация обязывает писать многочисленные обработчики ошибок, чтобы минимизировать последствия конфликтов типизации. Статические типы избавляют от необходимости писать запутанные обработчики. |  |
| Изменение сигнатуры функций неизбежно влечет рефакторинг: поменяли тип одного из параметров — будьте добры, поменяйте все вызовы этой функции. С динамической типизацией пришлось бы искать все вызовы функции самостоятельно и надеяться, что ни один из них не был пропущен. Статическая типизация на стадии разработки сигнализирует о том, что тип параметра не соответствует сигнатуре, и что нужно скорректировать вызов функции в определенном месте. |  |

Решение о том, стоит ли использовать статическую типизацию в коде или нет, всегда остается за разработчиком. Мы рекомендуем руководствоваться следующими идеями.

Если ваша программа:

* + важна для бизнеса;
  + сложна и имеет много динамических деталей;
  + поддерживается большой командой, которой необходимо быстро и точно понимать код;
  + скорее всего будет меняться по мере развития ваших потребностей.

То стоит отдать предпочтение **статическим типам**— они сыграют вам на руку.

Если же:

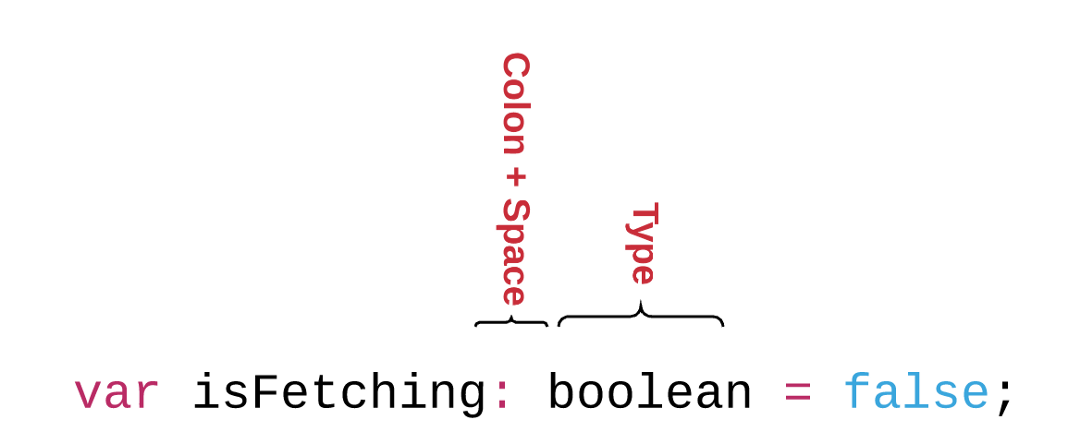
* + вам нужно создать работающий прототип как можно быстрее;
  + программа небольшая и простая;
  + код не задуман для дальнейшего использования;
  + вы — единственный разработчик.

То тогда вы можете без ущерба отказаться от использования инструментов для статической типизации.

**Flow**

Для того чтобы [Flow](https://flow.org/en/docs/getting-started/" \t "[object Object]) начал анализировать ваш код, необходимо в начале файла добавить комментарий /\* @flow \*/.

Конструкция, используемая для указания типа проста: абсолютно такая же используется и в *TypeScript*. Обратите внимание: тип указывается между названием переменной и присвоением значения.



Добавление типов к примитивам будет выглядеть следующим образом:

**var** isFetching: **boolean** = **false**;

**var** luckyNumber: number = **10**;

**var** notSoLuckyNumber: number = **NaN**;

**var** myName: string = 'Preethi';

**var** data: **null** = **null**;

**var** data: **void** = **undefined**;

**Обратите внимание!** В данном случае название типа не дублирует зарезервированное слово.

С объектами уже не всё так просто:

Самый простой вариант показать, что переменная будет объектом — использование слова Object. В этом случае *Flow* будет проверять, является ли ваша переменная объектом или нет, при этом объект внутри может быть любым.

**var** someObject: Object = {};

someObject.name = {};

someObject.name.first = 'Preethi';

someObject.age = **26**;

Если вы всё-таки хотите проверять содержимое объекта, есть возможность описать его структуру. Другими словами, описать названия полей, которые должен содержать ваш объект, и их типы.

**var** aboutMe: { name: string, age: number } = {

name: 'Preethi',

age: **26**,

};

Для типизации массивов используется Array, где T — тип элементов массива. Получается, что записать в один массив элементы разных типов уже не получится.

**var** messages: Array<string> = ['hello', 'world', '!'];

Однако здесь не обошлось без лазейки: можно типизировать переменную с помощью any. Any говорит о том, что переменная может быть любого типа. Иногда это полезно, но не стоит злоупотреблять этим типом, потому что он сводит на нет смысл статической типизации.

Для работы с функциями тоже используется типизация: указываются типы для входных параметров функции и возвращаемого значения.

**var** calculateArea = (radius: number): number => {

**return** **3.14** \* radius \* radius

};

Ну и, конечно же, есть возможность описать свой собственный тип:

type PaymentMethod = {

id: number,

name: string,

limit: number,

};

Очень часто нам приходится работать с однотипными объектами, и чтобы было легче указывать их тип, можно создать кастомный тип и использовать его.

**var** myPaypal: PaymentMethod = {

id: **123456**,

name: 'Preethi Paypal',

limit: **10000**,

};

Также, в *Flow* существует тип Maybe. Он существует для того, чтобы показать, что переменная может быть одним из трех типов: T (который установили сами), null или void.

**var** message: ?string = **null**;

message может быть либо строкой, либо null, либо undefined.

А если мы поместим вопросительный знак не перед самим типом, а перед двоеточием — мы покажем, что поле необязательно.

type PaymentMethod = {

id: number,

name: string,

limit?: number,

};

Если мы создадим объект {id: 123, name “John”} и укажем тип PaymentMethod, то ошибки не будет.

Если вы еще не успели запутаться, то вот ещё одна возможность описания типа:

type GenericObject<T> = { key: T };

**var** numberT: GenericObject<number> = { key: **123** };

**var** stringT: GenericObject<string> = { key: "Preethi" };

**var** arrayT: GenericObject<Array<number>> = { key: [**1**, **2**, **3**] }

Мы уже видели похожий механизм немного ранее, когда рассматривали типизацию массивов. Такие типы, как GenericObject называют *параметризованными* или *обобщенными* (Generic). Каждый раз при создании объекта данного типа мы должны указывать параметр T, который и будет использоваться для типизации (в данном случае поля key).

Ну и последнее: **непересекающиеся множества**.

Иногда случается так, что мы не можем предсказать, какой тип мы должны использовать. Представьте, что нам необходимо реализовать страницу оплаты для магазина резиновых уточек. Пользователю на выбор может быть представлено несколько способов оплаты, и нам необходимо их различать для того, чтобы отправлять информацию на сервер. Например, есть следующие опции:

type Paypal = { id: number, type: 'Paypal'};

type CreditCard = { id: number, type: 'CreditCard'};

В остальном системе не важно, какой способ оплаты выбрал пользователь. Чтобы не писать одинаковые обработчики на каждый тип, можно описать абстрактный PaymentMethod, который будет принимать форму одного из методов в зависимости от ситуации.

type PaymentMethod = Paypal | CreditCard;

То есть платежный метод может быть или *Paypal*, или *CreditCard*.

**TypeScript**

Как и для работы с *Flow*, необходимо явно указывать, какие файлы должны проверяться, с той лишь разницей, что для *TypeScript* мы создаем файлы с расширением *.TS*. https://flow.org/en/docs/getting-started/

Поскольку *TypeScript* является надмножеством *JavaScript,* то корректно написанный на *JavaScript* является также кодом *TypeScript.* Основное преимущество *TypeScript* заключается в том, что он может выделять неожиданное поведение в вашем коде, снижая вероятность ошибок.

Так как *TypeScript* основан на *JavaScript,* во многих случаях он будет генерировать для вас типы, если вы не указали их явно. Например, при создании переменной и присвоении ей определенного значения *TypeScript* будет использовать это значение в качестве своего типа.

**let** helloWorld = "Hello World";

// ^ = let helloWorld: string

Помимо примитивных типов, уже существующих в *JavaScript:* boolean, bigint, null, number, string, symbol, object, и undefined, *TypeScript* имеет уже знакомые нам типы any (разрешает использование любого значения) и void (тип для функций, которые ничего не возвращают (у них нет return)).

Так же, как и *Flow, Typescript* позволяет создавать псевдонимы типов, но помимо них здесь также есть интерфейсы.

type Container<T> = { value: T };

**interface** User {

name: string;

id: number;

}

Использование интерфейса:

**const** user: User = {

name: "Hayes",

id: **0**,

};

Если объект не будет соответствовать интерфейсу, *TypeScript* предупредит вас:

**interface** User {

name: string;

id: number;

}

**const** user: User = {

username: "Hayes",

Type '{ username: string; id: number; }' is not assignable to type 'User'.

Object literal may only specify known properties, and 'username' does not exist **in** type 'User'.

id: **0**,

};

Почти все функции interface доступны в type, ключевое различие заключается в том, что тип нельзя повторно открыть для добавления новых свойств по сравнению с интерфейсом, который всегда можно расширить. Считается, что предпочтительнее использование интерфейсов.

Кроме того, *TypeScript* имеет специальные операторы, которые помогают преобразовывать тип переменной. Ключевое слово asas и оператор <>.

**let** str: any = 'Текстовая переменная'

**let** strLength = (<string>str).length

и

**let** str: any = 'Текстовая переменная'

**let** strLength = (str as string).length

Массив можно обозначить с помощью [], при этом как и *Flow, TypeScript* позволяет типизировать массивы с помощью обобщенного типа, но помимо этого поддерживает мультитипные массивы:

**let** stringsAndNumbers: (string | number)[] = ['Age', **20**]

*TypeScript* поддерживает многомерные массивы: можно сохранять массивы в других массивах. Создать такой массив можно через множественный оператор  [].

**let** numbersArray: number[][] = [[**1**,**2**,**3**,**4**,**5**], [**6**,**7**,**8**,**9**,**10**]]

Также в *TypeScript* существует тип enum (перечисления), который позволяет создавать именованные константы:

**enum** State{

Playing = **0**,

Paused = **1**,

Stopped = **2**

}

Важным отличием от *JavaScript* является и то, как происходит работа с функциями: компилятор проверяет корректность передаваемых параметров (их количество и типы).

**function** add(num1: number, num2: number): number {

**return** num1 + num2

}

add(**2**, **5**)

add(**1**) // "Error too few parameters"

add(**5**, '2') // "Error the second argument must be type number"

Однако имеется возможность сделать параметры необязательными с помощью оператора ?.

**function** add(num1: number, num2?: number)

В этом случае, если вы передадите в функцию только один параметр, ошибки не будет. Для аргументов функции можно назначать значение по умолчанию. Это значение будет присваиваться аргументу тогда, когда при вызове функции значение самого аргумента не было передано.

**function** printName(firstName: string, lastName: string = 'Tanner') {  
console.log(`Firstname: ${firstName}, Lastname: ${lastName}`); }

Еще одной особенностью *TypeScript* является наличие модификаторов доступа, которые управляют доступностью членов класса.

*TypeScript* поддерживает 3 модификатора:

* *Public — э*лементы с этим модификатором доступны отовсюду без каких-либо ограничений. Этот модификатор установлен по умолчанию.
* *Private — э*лементы с этим модификатором доступны только из того класса, в котором они определены.
* *Protected — э*лементы с этим модификатором доступны из класса, в котором они определены, и в подклассах/производных классах.

*TypeScript* богат функциональностью, которой не хватает в *JS,* и мы перечислили только некоторые особенности.

Рекомендуем почитать [официальную документацию](https://www.typescriptlang.org/docs), чтобы еще глубже изучить этот вопрос. https://www.typescriptlang.org/docs/

Мы с вами рассмотрели два инструмента, которые имеют похожий синтаксис и выполняют одну задачу. Разница между этими инструментами концептуальная: *Flow* — это анализатор, который контролирует типизацию, а *TypeScript* — язык программирования, компилирующийся в *JS*.

Давайте поговорим, как можно подключить *TypeScript* в ваш проект и использовать его. Как обычно, все начинается с установки загрузчика и компилятора для *TS:*

npm install --save-dev typescript ts-loader

После этого в нашем проекте должен появиться tsconfig.json и index.ts. Далее мы исправляем файл webpack.config.js.

Во-первых, с этого момента точкой входа в наше приложение становится файл index.ts. Во вторых, мы подключаем загрузчик ts-loader.

{

test: /\.tsx?$/,

use: 'ts-loader',

exclude: /node\_modules/,

},

Давайте попробуем повторить это на практике.

<https://lms.skillfactory.ru/courses/course-v1:Skillfactory+FR+2020/courseware/4277fd5e8f184b7ebf2eb5b830e09fdd/900019ad62c94284969a6672e8010e4e/?child=first>

## 13.8 Дополнительные инструменты для качества кода

https://lms.skillfactory.ru/courses/course-v1:Skillfactory+FR+2020/courseware/4277fd5e8f184b7ebf2eb5b830e09fdd/900019ad62c94284969a6672e8010e4e/?child=first

Все мы иногда допускаем ошибки. Одни вполне серьёзные, а другие мы даже не замечаем. Однако, даже такие, казалось бы, ерундовые ошибки имеют последствия. Вчерашний небрежно написанный или чересчур запутанный код может заставить вас и ваших коллег через пару недель гадать на кофейной гуще, что же имел в виду автор, и выдумывать изощренные костыли.

Одним из оберегов от подобного рода проблем является **код-ревью**.

**Code-review**— процедура проверки написанного вами кода вашими коллегами.

Ведь одна голова хорошо, а две лучше. И как гласит старинная мудрость: семь раз проверь, один — *merge*.

Но нет ничего более субъективного, чем красота. Особенно, когда речь идет о красоте кода. Для того, чтобы каждое *code-review* не заканчивалось битвой между разработчиками, существует такое понятие, как *code-style*. Это своего рода мирное соглашение, где описаны договоренности о том, какой код считается красивым в контексте данного приложения.

**Code-style** — набор соглашений и правил, используемых при написании кода на некотором языке программирования.

Придерживаться *code-style* — дело трудозатратное: коварные ошибки, нарушающие чистоту кода, просто напросто не цепляют глаз, и их легко пропустить.

Здесь нам на помощь приходят специальные механизмы, которые распознают такого рода ошибки и будут учитывать тот самый *code-style*, которого придерживаются разработчики. Инструменты, предоставляющие нам эти механизмы, называются **линтерами**.

**Linter** — программа, которая проверяет исходный код на соответствие стандартам, опираясь на определенный набор правил.

Их можно настроить на свой вкус и запускать перед тем, как отправлять свой код на суд коллегам. Линтер пробегает по всему проекту и отыскивает все те огрехи, которые трудно найти. Помимо этого можно подключить линтер в своей среде разработки, и тогда ошибки будут подсвечиваться по ходу того, как вы будете писать код. Естественно, для каждого языка программирования найдется хотя бы одно решение, но мы с вами сегодня рассмотрим инструменты для улучшения кода на *HTML*, *CSS* и *JS*.

**ESLint**

**ESLint**— это инструмент для выявления шаблонов, обнаруженных в коде *ECMAScript* / *JavaScript*, и составления отчетов о них с целью сделать код более согласованным и избежать ошибок.

Для начала работы с линтером установим зависимости:

npm install eslint --save-dev

Затем необходимо добавить файл конфигурации:

npx eslint --init

После чего можно запустить линтер для любого файла из каталога следующим образом:

npx eslint yourfile.js

Для удобства можно добавить этот скрипт в package.json, чтобы в дальнейшем пользоваться npm run lint.

После запуска у eslint --init в вашем каталоге появится файл .eslintrc.js. В нем вы увидите некоторые правила, настроенные следующим образом:

{

"rules": {

"semi": ["error", "always"],

"quotes": ["error", "double"]

}

}

Имена "semi" и  "quotes" — это названия правил в *ESLint*. Первое значение — это уровень ошибки правила и может быть одним из следующих значений:

* + "off" или 0 — выключить правило
  + "warn" или 1— включить правило как предупреждение (не влияет на код выхода)
  + "error" или 2 — включить правило как ошибку (код выхода будет 1)

Три уровня ошибок позволяют детально контролировать то, как *ESLint* применяет правила.

Кроме того, файл eslintrc.js будет содержать строку, которая по умолчанию включает все рекомендованные правила (в [документации](https://eslint.org/docs/rules/) они обозначены галочкой) .

{"extends": "eslint:recommended"}

Ну и конечно, если вы не нашли нужное правило из списка уже существующих, всегда можно написать свое.

*ESLint* разработан так, чтобы быть полностью настраиваемым, что означает, что вы можете отключить каждое правило и запускать только с базовой проверкой синтаксиса или смешивать и сопоставлять связанные правила и свои собственные правила, чтобы сделать *ESLint* идеальным для вашего проекта. Существует два основных способа настройки *ESLint*:

* 1. **Комментарии к конфигурации** — используйте комментарии *JavaScript* для вставки информации о конфигурации непосредственно в файл.
  2. **Файлы конфигурации** — используйте файл *JavaScript*, *JSON* или *YAML*, чтобы указать информацию о конфигурации для всего каталога и всех его подкаталогов. Это может быть в форме .eslintrc.js файла или eslintConfig поля в package.json файле, оба из которых *ESLint* будет искать и читать автоматически, или вы можете указать файл конфигурации в командной строке.

Можно настроить несколько элементов информации:

* Среды — среды, для которых предназначен ваш сценарий. Каждая среда имеет определенный набор предопределенных глобальных переменных.
* *Globals* — дополнительные глобальные переменные, к которым ваш скрипт обращается во время выполнения.
* Правила — какие правила включены, и на каком уровне ошибки.

Все эти параметры дают вам детальный контроль над тем, как *ESLint* обрабатывает ваш код.

**Попробуем настроить**

Снова начинаем с установки зависимостей. На этот раз нам понадобится eslint-loader, который нужно добавить в список загрузчиков, через которые будут проходить *JS*файлы.

## ****CSS lint****

Аналогичный инструмент есть и для CSS. Давайте посмотрим на stylelint-webpack-plugin.

Npm i stylelint stylelint-webpack-plugin --save-dev

А далее по аналогии с установкой других плагинов, мы меняем конфигурационный файл.

**const** StylelintPlugin = require('stylelint-webpack-plugin');

module.exports = {

// ...

plugins: [**new** StylelintPlugin(options)],

// ...

};

Теперь при запуске сборки проекта все CSS-файлы будут прогоняться через линтер, и сборка будет успешно завершена только в том случае, если в коде не будет стилевых ошибок.

Хоть модуль и получился довольно объемным, мы надеемся, что это время вы провели с пользой. Вы узнали о таких инструментах, как сборщики проектов, и попробовали самостоятельно настроить *Webpack*. Также, теперь вы знаете, как добавить в *JavaScript* статическую типизацию и подключить линтеры. Эти знания одинаково пригодятся вам и во время работы над собственными проектами, и в командной разработке.

Теперь вы:

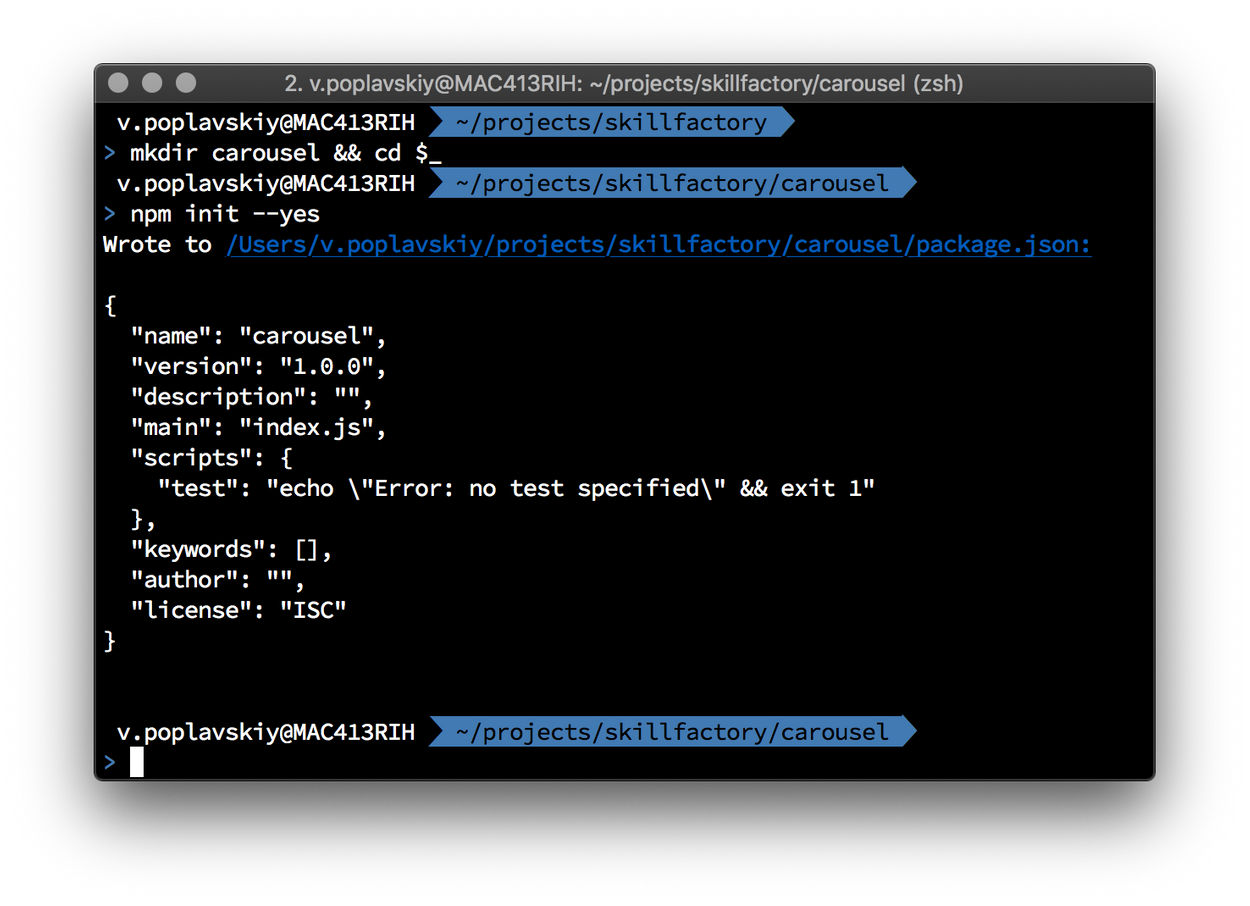
* Знаете в чем разница между *SPA* и *MPA;*
* Понимаете зачем нужен *Webpack* и можете его сконфигурировать;
* Умеете добавлять статическую типизацию в свой код на *JavaScript;*
* Представляете, что такое минификаторы и шаблонизаторы, и как подключить их в свой проект.

УСТАНОВКА ВЭБПАКА В ПРОЕКТ КАРУСЕЛЬ. МОДУЛЬ 20.3

## ****Создание проекта с помощью библиотеки****

Давайте создадим новый проект с помощью npm и используем в нём библиотеку [Flickity](https://flickity.metafizzy.co/" \t "[object Object]) для каруселей.

Создайте новую папку carousel, перейдите в неё и используйте npm init --yes для создания npm-проекта, то есть для генерации дефолтного файла package.json, в котором перечислены все зависимости (то есть используемые npm-пакеты) этого проекта.



Дальше откроем наш проект с помощью IDE или редактора кода и создадим небольшую структуру проекта, то есть папку src, а в ней файлы index.html и index.js.

Затем инициализируем новый git-репозиторий командой git init и сделаем первый commit c названием "Initial commit" командами git add . и git commit -m "Initial commit".

## https://lms.skillfactory.ru/assets/courseware/v1/c4b60fa1c8eb03c96a72564642cf2891/asset-v1:Skillfactory+FR+2020+type@asset+block/D2.2.2.png

## ****Установка webpack****

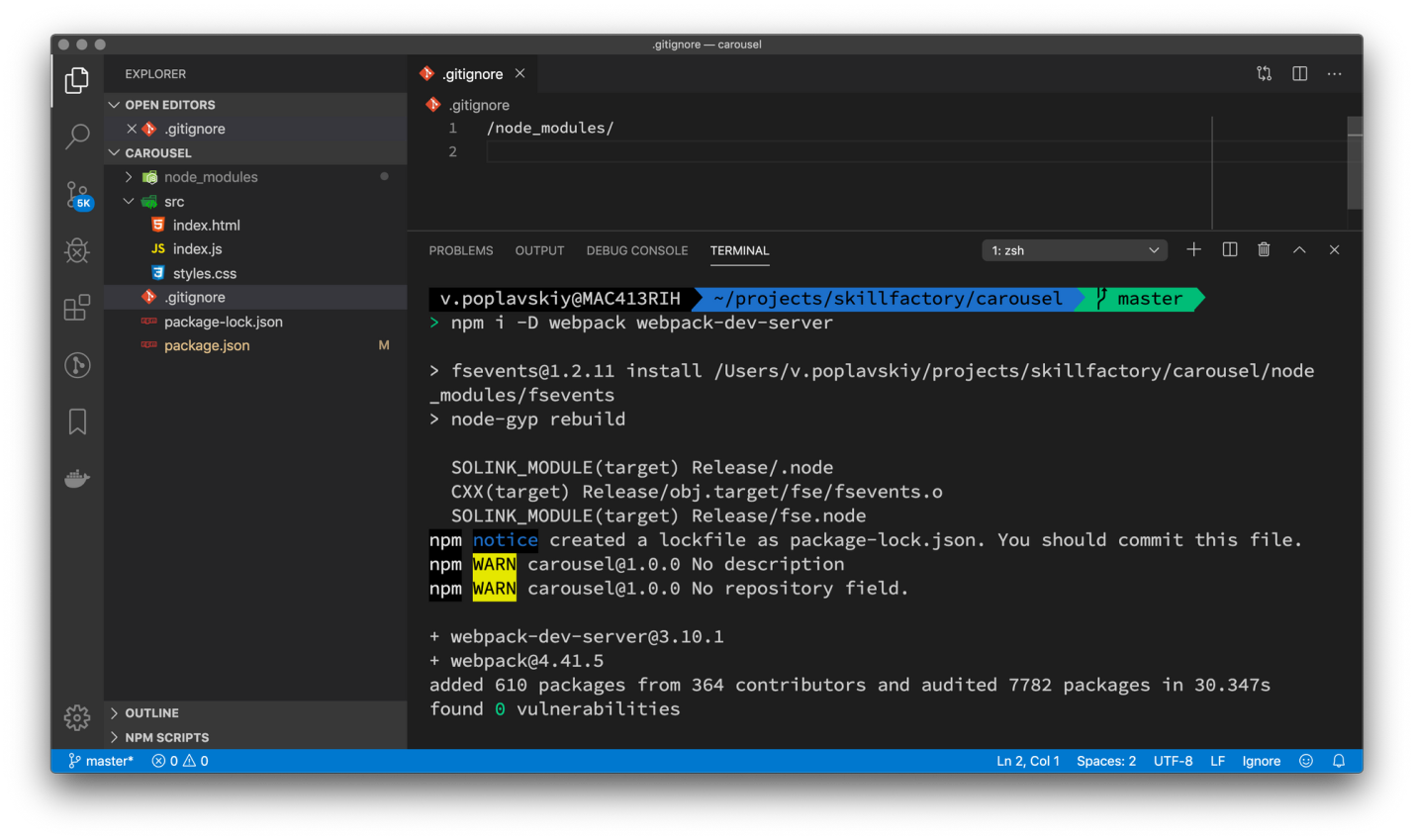
Теперь нам нужно собрать наш проект с помощью Webpack и проверить, работает ли он.

Перед установкой библиотек мы должны создать файл с названием .gitignore  и записать в него одну строчку: node\_modules, чтобы git не учитывал сотни тысяч файлов сторонних библиотек, которые будут лежать в /node\_modules/.

Напомним, что Webpack является библиотекой для Node.js. Он есть в npm, поэтому мы можем установить его с помощью команды npm i -D webpack.

Флаг -D — это сокращённо --save-dev. Он нужен, чтобы указать, что сам Webpack не попадёт в проект и не будет использоваться в браузере у конечного пользователя нашего приложения, то есть это[development dependency](https://docs.npmjs.com/specifying-dependencies-and-devdependencies-in-a-package-json-file), которая используется только для сборки.

Также нам понадобится npm i -D webpack-dev-server webpack-cli, чтобы запустить наше приложение и посмотреть, работает ли оно.



Теперь создадим в корне проекта файл webpack.config.js и скопируем из [документации](https://webpack.js.org/guides/development/#using-webpack-dev-server) Webpack простейший конфиг для dev-server'а:

**const** path = require('path');

**const** HtmlWebpackPlugin = require('html-webpack-plugin');

**const** { CleanWebpackPlugin } = require('clean-webpack-plugin');

module.exports = {

mode: 'development',

entry: {

app: './src/index.js',

},

devtool: 'inline-source-map',

devServer: {

contentBase: './dist',

},

plugins: [

**new** CleanWebpackPlugin(),

**new** HtmlWebpackPlugin({

template: 'src/index.html',

}),

],

output: {

filename: '[name].bundle.js',

path: path.resolve(\_\_dirname, 'dist'),

},

};

И установим ещё парочку плагинов для его работоспособности.

npm i -D html-webpack-plugin clean-webpack-plugin

## ****Проверка работоспособности сборщика****

Теперь можно в файле index.js написать проверочный код, например, какой-нибудь alert, а в файле index.html задать стандартную разметку (например, при использовании автодополнения IDE в VSCode достаточно напечатать ! и нажать tab).

**index.html**

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Hello world!</title>

</head>

<body>

</body>

</html>

**index.js**

alert('Hello world!');

Ещё нам нужно задать в package.json скрипты для запуска сборки. Находим поле scripts и дописываем туда команды из документации Webpack:

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1",

"start": "webpack-dev-server --open",

"build": "webpack"

},

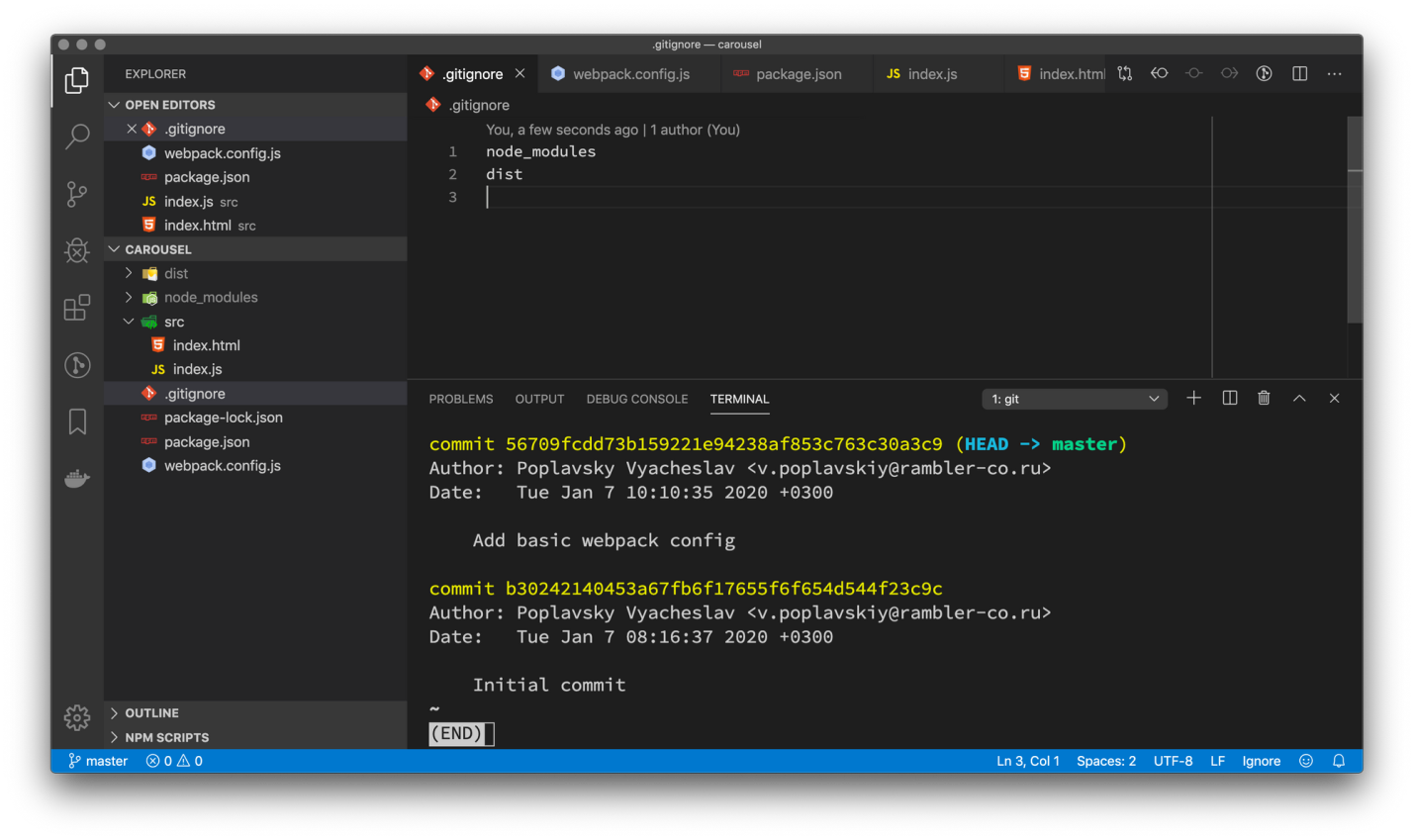
Теперь при запуске команды npm start у нас должен запуститься webpack-dev-server и открыться браузер со страницей index.html, а при запуске npm run build проект должен собраться в папку /dist.

Проверьте, всё ли получилось, и сработал ли alert('hello world') на странице.

Если всё хорошо, нужно добавить папку dist в .gitignore и закоммитить наши изменения:

git add .

git commit -m "Add basic webpack config"



## ****Подключение flickity****

Теперь всё готово для подключения карусели на страницу.

Устанавливаем flickity через npm, для этого находим [страницу библиотеки](https://www.npmjs.com/package/flickity) в поиске npm. <https://www.npmjs.com/package/flickity>

И находим в разделе ***Install*** команду для установки:

npm i flickity

Обратите внимание, что флаг -D писать не надо, так как эта библиотека попадёт к пользователю, то есть она является обычной dependency.

Можно скопировать из документации пример использования библиотеки и дописать наш **index.html**:

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title>Hello world!</title>

</head>

<body>

<div class="carousel">

<div class="carousel-cell">...</div>

<div class="carousel-cell">...</div>

<div class="carousel-cell">...</div>

</div>

</body>

</html>

Теперь осталось только импортировать нашу библиотеку и использовать её согласно документации:

**import** Flickity from 'flickity';

**const** flickity = **new** Flickity('.carousel');

Запускаем npm start и видим, что что-то не так:



Мы забыли подключить стили. Это можно сделать через CDN, указанный в документации по flickity, либо добавить в Webpack возможность собирать CSS.

Ищем в [документации](https://webpack.js.org/guides/asset-management/#loading-css) по Webpack, как это сделать.

module: {

rules: [

{

test: /\.css$/,

use: ['style-loader', 'css-loader'],

},

],

},

Устанавливаем style-loader и css-loader:

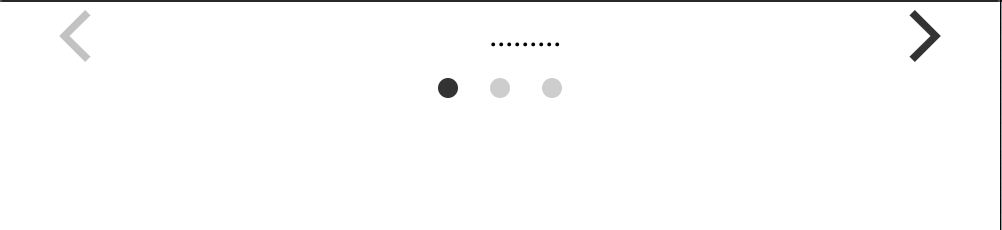
npm install -D style-loader css-loader

Добавляем импорт CSS в index.js:

import Flickity from 'flickity';

import 'flickity/dist/flickity.min.css';

const flickity = new Flickity('.carousel');



Готово! Стили карусели подключились, теперь осталось добавить наши стили, чтобы карусель выглядела лучше. Создаем файл style.css рядом c index.js . Теперь, используя dev-server, мы можем менять стили и сразу видеть результат. Импортим наши стили рядом со стилями библиотеки:

**import** Flickity from 'flickity';

**import** 'flickity/dist/flickity.min.css';

**import** './style.css';

**const** flickity = **new** Flickity('.carousel');

И зададим какие-нибудь стили слайдам галереи:

body {

**margin**: **0**;

}

**.carousel-cell** {

**width**: **100**%;

**height**: **400px**;

**font-size**: **60px**;

**line-height**: **380px**;

**text-align**: **center**;

**border-right**: **5px** **solid** blue;

**border-left**: **5px** **solid** blue;

**background-color**: red;

**color**: white;

}

Теперь карусель выглядит чуть более похожей на карусель:

