[[](https://habr.com/ru/users/Psychosynthesis/)Psychosynthesis](https://habr.com/ru/users/Psychosynthesis/" \o "Автор публикации)4 марта 2019 в 19:20

Как подступиться к fullstack-разработке сегодня, если ты проспал десять лет

* [Разработка веб-сайтов](https://habr.com/ru/hub/webdev/),
* [Программирование](https://habr.com/ru/hub/programming/),
* [Софт](https://habr.com/ru/hub/soft/)
* [Recovery Mode](https://habr.com/ru/info/help/karma/#recovery)

  
  
Привет, Хабр! Несколько месяцев назад у меня остро встал вопрос смены профиля деятельности и я обнаружил, что для претендента на вакансию web-разработчика сейчас недостаточно навыков десятилетней давности (какая неожиданность!). Пришлось срочно актуализировать свои знания. Заодно я решил составить шпаргалку с описанием большинства современных технологий, чтобы в случае чего кидать жаждущим новых знаний линк на эту статью, да и самому не забывать.

В качестве вступления...

**Зачем это тут?** Вполне вероятно, для многих пользователей Хабра всё описанное в статье покажется очевидным. Более того, какие-то аспекты, разумеется уже были описаны более подробно, да не один раз. Однако человеку, который знает только основы (HTML/CSS/JS), всё происходящее, например, в современном JS кажется просто хаосом, в котором решительно ничего не понятно и не ясно даже с какой темы начать изучать вопрос. Когда пытаешься такому человеку что-либо рассказать, упираешься в необходимость рассказывать массу вещей из разных областей, что превращает повествование в кашу.  
  
*Я не претендую на глубокое знание всех описываемых технологий, поэтому буду рад любым дополнениям и комментариям от специалистов — хотелось бы сделать действительно качественный обзор.*

Disclaimer

Чтобы не слишком раздувать статью, минимум внимания уделено фундаментальным понятиям, вроде архитектурных решений или паттернов программирования (которые вообще-то неплохо было бы знать априори). Также не будут рассматриваться отдельные обширные вопросы, вроде веб-серверов или особенностей вёрстки на CSS3 — иначе это всё превратится не в обзор, а в учебник.  
  
Также, тут совсем ничего нет про функциональные языки: Scala, Erlang, Haskell, etc.

Итак, описывая базовые вопросы я буду исходить из того, что читатель только-только начинает свой путь в чудном мире fullstack, поэтому для понимания остального материала я настоятельно рекомендую прогуглить и вдумчиво прочесть информацию по нижеприведённому списку. Даже если сразу понять что к чему вам не удастся, советую прочитанное запомнить и потом, в случае чего, вернуться и вот тогда всё встанет на свои места.

Базовые понятия

* В первую очередь, актуализируйте свои знания о современных стандартах, используемых для вёрстки (HTML5, CSS3 и вот это всё). Также полезно будет прочесть про эволюцию ECMA Script (JS, на самом деле, это реализация именно этого стандарта), и освежить свои знания про JSON и JWT.
* **Объектно-ориентированный подход в программировании**. Да-да! Многие, кто, вроде бы, понимает о чём речь, саму идею не до конца осознают, хотя она и появилась лет 50 назад. Поэтому желательно ещё раз ознакомиться, если хоть чуть-чуть сомневаетесь.
* **Системы контроля версий**. Это абсолютный вин, знать их сегодня просто необходимо всем, кто работает в сфере разработки любого ПО. Даже если работаешь один системы контроля версий очень полезны, а уж в команде без них просто ад. Раньше для этого использовали, в основном, SVN или CVS, просто потому что серьёзных конкурентов у них почти не было. Сейчас стандарт де-факто это GIT (спасибо Линусу Торвальдсу). Я рекомендую начать с изучения консольной версии (даже для Windows), это займёт от силы пару дней, но когда придёт понимание, все фишки (типа git flow) вы поймёте очень быстро, а там уже и GUI-клиент какой-нибудь можно будет поставить. И уж совсем крутым можно стать, например, если освоить git hooks и настроить отправку данных на боевой сервер при коммите в мастер-ветку репозитория проекта.
* Концепция **MVC** (не будет преувеличением сказать, что практически во всём современном вебе ПО работает по этому принципу, поэтому читать точно придётся).
* Концепция **RestAPI**.
* Виды, особенности и различия баз данных и основы SQL (я не говорю что это надо прямо учить, но представление иметь однозначно надо, вплоть до составления хотя бы простейших запросов). Почитать про NoSQL-базы данных, в частности MongoDB и Redis.
* Методологии **БЭМ**. Я не сторонник такого подхода, но многие работают с его использованием, чтобы понимать о чём речь желательно почитать хотя бы вкратце.
* Методологии разработки **Agile** и **Scrum** (там ничего особо сложного, скорее для общего развития).
* Концепция **Single Page Application** (SPA) — применима не всегда, но почему-то всё чаще используется даже для очень объёмных проектов. Читать определённо.
* Почитайте про **Web Sockets** — это технология, которая позволяет создавать интерактивное соединение между клиентом (браузером) и сервером для обмена сообщениями в режиме реального времени. Веб-сокеты, в отличие от HTTP, позволяют работать с двунаправленным потоком данных. Предлагаю смотреть на это как на AJAX нового поколения. Основная плюшка — это отсутствие необходимости постоянно запрашивать новые данные с сервера. При необходимости сервер сам отправит данные, а браузер их получит.
* Ещё один стандарт который, вероятно, значительно изменит фронтэнд-разработку уже в ближайшем будущем — **Web Components**. Очень мощный инструмент и он вам определённо пригодится, хотя в данной статье я не буду уделять ему внимания.
* **XPath** — язык запросов к DOM-документа. Маловероятно что он вам пригодится, так как в абсолютном большинстве случаев CSS-селекторы удобнее, но знать о его существовании полезно. Сейчас используется в основном в области тестирования или для парсинга больших объёмов данных.
* Одно из ключевых отличий, по сравнению с разработкой десяток лет назад — сейчас практически везде принято использовать т.н. **пакетные менеджеры** (менеджеры зависимостей), которые пришли из мира UNIX. О некоторых из них я расскажу более подробно далее. Основная идея в том, чтобы снять с плеч программиста заботу о соблюдении зависимостей и обновлении используемых библиотек и фреймворков. Если раньше для начала проекта требовалось руками копировать в нужные места все нужные файлы и библиотеки, прописать все пути, проследить чтобы они были между собой совместимы и т.п., то сейчас всё это делается парой команд в консоли (да, без консоли до сих пор никуда, увы).
* **Прекомпиляторы**. Возможно звучит странно, так как мы говорим о интерпретируемых языках, но таки да, сплошь и рядом используются отдельные программы, позволяющие, например, собрать все используемые стили в один файл перед заливкой в продакшн, а также позволяющие использовать в своём коде всякие удобные штуки, которых нет в стандарте (например переменные в CSS). Особенно популярны прекомпиляторы во фронтэнде — так как для бэкэнда чаще всего используются полноценные ООП-языки, там прекомпиляторы не так распространены и применяются, в основном, для ускорения и кэширования (например PHP Zend OPcache).
* Для упрощения развёртывания разработки сейчас принято использовать т.н. **контейнеры** (Docker, Kubernetes). Это относительно новая концепция в современной разработке, рассматривать подробно её я не буду, однако представление иметь надо — технология очень мощная и однозначно полезная.
* **Тестирование** — отдельная большая тема. Чаще всего под этим подразумевается т.н. юнит-тестирование. Суть идеи легко понять на простом примере. Создавая тест вы пишете некую обёртку, которая выполнит одну из ваших функций и проверит ожидаемый результат. Затем, когда вы поменяете что-либо в другой функции, от которой зависит первая функция, чтобы проверить, что изменение ничего не сломало — вам достаточно запустить созданные ранее тесты. Надеюсь понятно, насколько это может быть полезно в большом проекте.

Есть один важный аспект, который необходимо рассмотреть более подробно для предотвращения путаницы с понятиями «бэкэнд» и «фронтэнд», а также для более глубокого понимания происходящего в современной web-разработке. Это эволюция самого подхода к формированию HTML отображаемого в браузере. Хотя всё это формировалось, конечно, гораздо раньше чем десяток лет назад, поэтому, вроде бы, должно быть понятно и тому, кто лет десять за тенденциями не следил.

* Самый первый и самый простой вариант — веб-сервер парсит запрос и просто выдаёт HTML-файл, соответствующий этому запросу. Максимум, может использоваться какой-нибудь SSI. По сути это просто классический статический сайт, а бэкэндом в данном случае выступает сам веб-сервер. Очевидно, для проекта в котором может быть множество страниц такой подход катастрофически неудобен — если нужно что-либо изменить, правки придётся вносить во всех файлах. Также очевидно, что ничего кроме статических сайтов тут реализовать практически невозможно. Сейчас это используется максимум для каких-нибудь совсем простых лендингов, и то, разработка фронтэнда для них всё равно сейчас ведётся, в основном, с помощью современных js-фреймворков, а их применение в абсолютном большинстве случаев практикуется посредством функциональности Node.js.
* Развитие статического варианта — использование интерпретируемых языков на стороне сервера. Тут появилась возможность писать шаблоны на одном из интерпретируемых языков, в которые подставляются данные в зависимости от запроса, а также стало возможно храненить данные не файлах, а в БД. Можно считать, что с этого момента появляется чёткое разделение:
  1. Собственно сам веб-сервер. Сейчас это уже вовсе не обязательно монолитное приложение, например, распространённый вариант — когда Nginx выступает в качестве балансировщика нагрузки, а сами запросы обрабатывает уже Apache. Хотя вариантов может быть множество (спасибо за уточнение [prijutme4ty](https://habr.com/ru/users/prijutme4ty/)).
  2. Язык и фрейморк на этом языке, которые используются для обработки запроса и маршртизации по URL. Пожалуй, не будет ошибкой сказать, что в контексте fullstack-разработки под понятием «бэкэнд» как раз данный слой чаще всего и подразумевается, хотя строго говоря, всё серверное ПО, обеспечивающее работу веб-сервера это, разумеется, тоже бэкэнд. Важно, что язык и фреймворк для бэкэнда могут быть практически любые. Например, в случае Ruby и Ruby on Rails в качестве шаблонизатора может использоваться встроенный ERB, в случае PHP и Laravel — в качестве шаблонизатора чаще всего используют Blade, ну и так далее.
  3. Код, который выполняется непосредственно на клиенте — то, что принято называть фронтэнд. Тут всё, фактически, сводится к JS.

Долгое время данный подход был основным и единственным в web-разработке, пока не начали развиваться облачные технологии и не появилась концепция SPA.

* Когда стали развиваться облачные вычисления, появились различные модели их использования (SaaS, PaaS, IaaS). Среди них хотелось бы отметить т.н. serverless-подход от AWS. Суть в том, что веб-сервер из предыдущего пункта заменяется на специальный фреймворк, посредством которого бэкэнд приложение взаимодействует с облаком. Облака развиваются семимильными шагами, появляется множество новых технологий и подходов к их использованию, поэтому описать всё кратко возможным не представляется. Если вам интересно данное направление, посмотреть большинство используемых сейчас облачных технологий с сортировкой по категориям можно тут: [landscape.cncf.io](https://landscape.cncf.io/) (спасибо [KonstantinSpb](https://habr.com/ru/users/konstantinspb/)).
* С развитием концепции SPA и появлением HTML-компонентов, которые рендерятся с помощью JS, встала проблема слишком долгого отображения — пока JavaScript загрузится и отрисуется у клиента, может пройти достаточно много времени. В связи с этим появился ещё один подход к генерации HTML, который исторически принято связывать именно со SPA-приложениями — Server Side Rendering (SSR). Суть в том, что мы можем выполнить первый рендер компонентов в HTML ещё на сервере, затем отдать это клиенту, вместе с тем же кодом, который сформировал этот HTML, и затем не передавать каждый раз целиком код приложения по сети, а просто отрисовывать нужные компоненты сразу в браузере, передавая лишь данные, необходимые для изменения состояния системы (то самое REST). Нужно понимать, что в некоторых случаях так могут не делать, разбив систему на независимые функциональные блоки (например, сгруппировав по формам, каждая из которых работает со своей областью системы), тогда это уже будет не SPA в классическом смысле. В таком случае рендеринг компонентов выполняется один раз перед отправкой с помощью какого-либо JS-шаблонизатора прямо на сервере — тогда, концептуально, это ничем не отличается от использования шаблонизатора на любом другом бэкэнд-языке, и, хотя, по сути, рендеринг из JS в HTML всё ещё будет происходить на стороне сервера, это уже не является SSR, потому что сие понятие **исторически** употребляется именно в контексте SPA. За прояснение сего момента спасибо товарищам [staticlab](https://habr.com/ru/users/staticlab/), [napa3um](https://habr.com/ru/users/napa3um/) и [justboris](https://habr.com/ru/users/justboris/).

Бэкэнд

Итак, в современном мире бэкэнд захватили несколько технологий. Встречаются, конечно всякие вариации и различная экзотика, однако полагаю не будет ошибкой сказать, что, навскидку, процентов 90 современного бэкэнда пишутся с помощью следующих инструментов:

* **Java**
* **.NET**
* **Python**
* **Node.js** (тут неоднозначно, но об этом дальше),
* **Ruby on Rails**
* И, конечно же, **PHP**

В данной статье мы не будем говорить про платформы .NET и Java — хотя и очевидно, что они в последнее время заняли очень обширную нишу, тем не менее, там свой отдельный зоопарк, описание которого заняло бы отдельную статью.  
  
Рассматривая оставшиеся технологии, когда речь заходит о том, в каком направлении проще всего найти работу, безусловными лидерами являются три языка — PHP, Node.js и Python.

Дабы не быть голословным вот результаты поиска вакансий на двух популярных ресурсах

Про Node.js и его экосистему я расскажу в конце статьи, поэтому сейчас, описывая бэкэнд основное внимание уделим Python и PHP.

Про Python...

… и используемые фреймворки от [artX89](https://habr.com/ru/users/artx89/):

Тем, кто по какой-либо причине еще его не знает, а возможно даже обходит его стороной, хочется посоветовать, все-таки, присмотреться к этому языку получше. Я сам очень долго не хотел его изучать, и причина была не как у многих («фу, да там все через отступы»), а в том, что приводя плюсы питона (да простят меня люди говорящие «пайтон»), обычно называют лаконичность языка.  
  
Часто приводят фразу в духе: «что на си занимает 100 строчек кода, в притоне это займет 10». Вот эта «простота» отпугивала, особенно после C++ и любимого С#. Казалось, что язык больше для «домохозяек» и людей не умеющих программировать на нормальных языках с нормальной типизацией, да и вообще язык только для скриптов. Я никогда так не ошибался! :) Язык очень мощный, невероятно удобный, который можно применять во многих областях. Я не знаю ни одного человека, перешедшего с PHP на питон, и который после этого опять хотел бы вернуться к PHP. Это я все к чему веду: питон отличный язык для того чтобы создавать на нем бэкэнд.  
  
Ну а теперь непосредственно про фреймворки… Для начала надо определиться: какой сервер нам нужен блокирующий или неблокирующий. У каждого из них есть плюсы и минусы. Неблокирующий сервер чаще всего используют для веб-сокетов, но на нем можно писать и обычные сайты, способные одновременно обрабатывать большое количество подключений. Но при работе с такими серверами надо помнить, что все подключения крутятся в одном «общем цикле» и блокировка этого «цикла» приведет к полной блокировке всех подключений, в связи с этим все используемые библиотеки должны быть не блокирующими. Среди неблокирующих фреймворков на сегодняшний день я бы советовал использовать aiohttp, есть и другие достаточно популярные такие как Tornado, но все они уступают aiohttp. Что же касается блокирующего фреймворка, то тут выделяется Django. Он подходит для людей, которые любят чтобы все было в одном флаконе и желательно сразу. Django уже «из коробки» включает в себя и шаблонизатор, и ORM, и многие другие удобные библиотеки. Но если Вы, как и, я любите выбирать библиотеки под задачу, тогда советую использовать Flask, а все остальные библиотеки уже настраивать под себя. Шаблонизатор чаще всего используют Jinja2, это достаточно удобный и распространённый шаблонизатор, с которым если и возникнут вопросы, то ответ можно будет нагуглить очень быстро. Что касается ORM, то в мире питона правит sqlalchemy. Это, как пишет создатель peewee (конкурента sqlalchemy): «SQLAlchemy is the gold standard for ORM in the Python world» — и я с ним полностью согласен. Это универсальный и мощный инструмент. Но за все надо платить, в данном случае приходится «платить» сложностью данного фреймворка. Хотя, начать с ним работать можно очень быстро, а вникать в сложности уже в процессе.  
  
Есть очень много и других полезных библиотек, полезных при работе с веб, таких как wtforms, beautifulsoup, Pillow и т.д, но тут все уже зависит от конкретного проекта и задач, которые стоят перед разработчиком.

Дополнение от [Stas911](https://habr.com/ru/users/stas911/):

Я бы еще добавил библиотеку requests (прямо очень мне понравилась). Для облачного serverless бэкенда стоит посмотреть на фреймворки Chalice и SAM. Ну и pipenv, black и flake8 наше все, конечно.

Что касается пакетного менеджера, самый используемый на данный момент в Python это Pip.  
  
Более подробно рассмотрим экосистему PHP, так как несмотря на тонны ненависти от сообщества, старичок живее всех живых и умирать пока не собирается.

PHP

В ходу сейчас 7-я версия языка. В первую очередь надо сказать, что различия, ломающие обратную совместимость, есть не только между 5-й и 7-й, но и, например между 7.0.х и 7.1.х (что, кстати, нарушает принципы «семантического версионирования», но всем плевать). В частности, одно из важных изменений, ломающих обратную совместимость в 7.1 — более строгая типизация параметров.

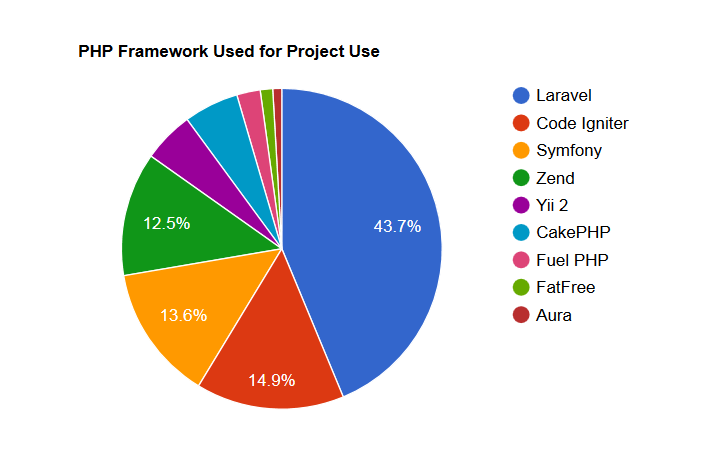
Пример функции, которая выполняется с предупреждением в PHP7.0 и выдает Fatal error в PHP7.1

Кроме таких перлов в язык добавилось чуть-чуть синтаксического сахара, а также были убраны некоторые встроенные функции… Была несколько изменена работа часто используемого цикла *foreach*… Пополнился список слов, которые нельзя использовать для именования собственных объектов (*resource, object, mixed, numeric, void, iterable*)… Ну и далее в таком же духе, ничего фундаментального, вроде бы.  
  
Если с PHP 4 или 5 вам уже приходилось иметь дело, вы достаточно быстро разберётесь.

Современные PHP-фреймворки

Беглый анализ предложений о работе позволяет выделить наиболее популярные сегодня фреймворки:

* **Laravel** — один из самых популярных фреймов. Ключевые фичи, доступные «из коробки»: шаблонизатор «Blade» (люблю его), Eloquent ORM для работы с БД и создания моделей (речь про модели из концепции MVC), механизм автоматической загрузки классов PHP без необходимости подключать файлы из определений в include, миграции (система управления версиями БД), есть свой формат пакетов, которые позволяют создавать и подключать модули Composer к приложению на Laravel. Многие дополнительные возможности уже доступны в виде таких модулей. Очень гибкий и, в целом, понятный инструмент. Будет хорошим стартом для дальнейшего изучения — при понимании используемых технологий, многие концепции других фреймворков будут понятнее. Также немаловажно, что данный фреймворк использует множество компонентов следующего фреймворка — Symfony, что тоже послужит хорошим подспорьем при дальнейшем расширении кругозора.
* **Symfony** — в качестве ключевых плюсов упоминаются универсальность, стабильность, однако тут достаточно высокий порог вхождения. Из коробки имеет два ORM-инструмента: Propel и Doctrine.
* **Zend** — самый высокий порог вхождения, взамен высокая стабильность и минимальные зависимости между частями проекта. Создан для разработки больших приложений корпоративного уровня (enterprise). Интересно, что Zend создан и поддерживается разработчиками PHP, а, например, виртуальная машина в интерпретаторе языка PHP именуется не иначе как Zend Engine.
* **Yii** — считается одним из наиболее производительных и при этом простых фреймворков, хотя однозначных подтверждений этому нет (Впрочем, есть интересное [сравнительное тестирование](https://github.com/kenjis/php-framework-benchmark), не сказать что оно всеобъемлющее, но скорость неплохо иллюстрирует. Спасибо [tnsaturday](https://habr.com/ru/users/tnsaturday/) ). У Yii огромное сообщество и высокая популярность. Имеется множество расширений, например для работы со Sphinx. Одна из «киллерфич» — генератор кода Gii, разумеется есть миграции БД.
* **CodeIgniter** — достаточно старый фреймворк, постепенно уходящий на задний план. Ключевые фичи это собственный гибкий шаблонизатор, шаблоны для работы с БД (очень похожи на синтаксис SQL), и мощные возможности кэширования на стороне сервера. Считается что это самый минималистичный и легкий фреймворк с одной из лучших документаций.
* От себя хотел бы также отметить ещё один фреймворк, ныне считающийся устаревшим, однако он хорош скоростью и минимализмом. Также, если начать изучение технологий с него, многие подходы в других фреймворках будут понятнее. Речь о **Kohana**(RIP) — (**Koseven**). Проект изначально отпочковался от CodeIgniter, и был написан для PHP5. На данный момент Kohana считается умершей, однако есть более-менее активный форк с поддержкой PHP7 — Koseven. В данном фреймворке имеется своя система ORM, модуль Minion для выполнения Cron-задач, есть своя система миграций БД, при желании несложно подключить расширения (например шаблонизатор Blade или модуль SCSS). По стилю чем-то напоминает Laravel.

Популярность PHP-фреймворков в 2018-м году по данным сайта Coderseye.com:  
  
  
  
Разумеется, это далеко не полный список, однако, как уже было сказано в начале — цель продемонстрировать основные тенденции. Выбрав любой из вышеперечисленных фреймворков для изучения — вы гарантированно найдёте себе работу. Исключение разве что Kohana, но и он поможет быстро понять Laravel, а также может пригодится для небольших проектов.

Также существует отдельный класс фреймворков...

Самым популярным пакетным менеджером для PHP сейчас является Composer.  
  
**Composer** — пакетный менеджер уровня приложений для языка программирования PHP, который предоставляет средства по управлению зависимостями в PHP-приложении. Простыми словами – это скрипт, написанный на PHP, который скачивает необходимые вам библиотеки, и автоматически формирует единственный специальный файл, подключив который, Вы подключите к проекту все скачанные библиотеки и фреймворки.  
  
На самом деле, насколько я понимаю возможности данного пакета, Composer можно использовать не только для проектов на PHP (например, с его помощью можно просто копировать файлы из указанного места, а затем запускать какой-нибудь Bower), однако в этом случае он превращается просто в обёртку для запуска скриптов и это явно не основное его применение.

Другие бэкэнд технологии

Ещё одна штука, часто используемая, например, для ускорения работы с реляционными БД — поисковые движки. Хочется упомянуть **Sphinx**. Это мощная система полнотекстового поиска, написанная на С++. Представляет из себя отдельное приложение, предоставляющее API для распространённых языков программирования (официально поддерживаются PHP, Python, Java; существуют реализованные сообществом API для Perl, Ruby, DotNET и C++) и интеграцию с существующими СУБД (MySQL, PostgreSQL). Для себя я делаю краткий перевод третьей версии документации Sphinx, если вам подобное интересно, отпишите, пожалуйста, в комментариях, по завершении я выложу его на хабр.  
  
*Хотелось бы добавить сюда ещё пару технологий, однако мне явно не хватает опыта в части бэка, чтобы выбрать о чём точно стоит упомянуть. Надеюсь на «помощь зала».*  
  
На этом с бэком более-менее всё ясно (не считая Node), поехали дальше.

Фронтэнд

Как я уже упоминал выше, сегодня очень популярны парадигмы, позволяющие объединить в себе как бэк, так и фронт, переложив на используемый фреймворк и задачу роутинга по URL внутри проекта и генерацию шаблона. В основном это всё касается Node.js и концепции SPA.  
  
Из сказанного в первой главе вытекает, что когда мы говорим о фронтэнде, под этим подразумевается код, который исполняется непосредственно в браузере пользователя. И тут, конечно, речь идёт только о JavaScript и его надмножествах.  
  
Я не зря упомянул вначале, что неплохо было бы почитать про ECMA Script, это необходимо для понимания того, почему во многих фреймворках используется, например TypeScript, а не привычный JS (кому-то пришло в голову, что без строгой типизации в JS жить ну никак нельзя, поэтому мы сейчас имеем то, что имеем). От себя скажу, что некоторые дополнительные фишки действительно очень удобны (например классы в ECMAScript 6, позволят писать заметно меньше кода, но мозг ломают знатно). Впрочем лично я являюсь приверженцем использования чистого JS, без всяких надмножеств, и без крайней необходимости новые возможности стараюсь не использовать.  
  
Несмотря на то, что в итоге всё свелось к JS, сам язык очень гибкий, способов реализовать то что хочется существует десятки, а сообществу постоянно нужно всё больше и больше новых фишек. Всё это привело к тому, что сия сфера сейчас похожа на настоящий хаос со множеством парадигм и направлений.  
  
На самом деле, про то что происходит сейчас в JS лучше всего написано вот [в этой статье](https://habr.com/ru/company/mailru/blog/340922/). Настоятельно рекомендую её прочесть, если ещё нет.  
  
Я буду более краток, поэтому пройдусь лишь по основным моментам, необходимым для понимания современного JS. Итак, наш отправной пункт это концепция JavaScript-модулей. Чтобы понять необходимость использования этой концепции достаточно рассмотреть простой пример.

Простой пример с jQuery, для наглядности

Проблема, на самом деле достаточно банальная, в некоторых случаях её можно обойти просто используя объекты, создающие свои области видимости, но в некоторых случаях она может серьёзно осложнить разработку сложного проекта.   
  
Кроме того, разработчикам хотелось максимальной унификации — чтобы модули для браузера и для сервера использовали одни и те же стандарты подключения к проекту. На самом деле, в действительно сложном проекте вопросов встаёт ещё много, например, проблема зависимостей между используемыми библиотеками. Всё это привело к развитию нескольких стандартов, позволяющих реализовать концепцию модулей — UMD, AMD и CommonJS.  
  
Наиболее часто используемая реализация на данный момент это формат CommonJS. В частности Node.js также использует эту реализацию. CommonJS реализует модули как специальный объект, внутри которого можно объявлять свои функции и переменные, а затем включать их в код в нужном месте с помощью ключевого слова *require*.  
  
Всё было бы здорово, только вот стандарты для браузеров разрабатываются заметно медленнее, чем растут амбиции разработчиков. Нативные модули впервые были описаны в ES6, а на текущий момент в браузерах только-только появился способ включать JS-файлы в качестве модулей, в то время как у пользователей ещё полным-полно старых браузеров, поэтому о нативной поддержке этой технологии говорить пока рано.  
  
Однако теперь, когда встретите в коде JS-проекта ключевые слова "*import*", "*export*", "*require*" — знайте, это как раз и есть реализация модулей.  
  
Но это ещё не всё. Модулей в проекте обычно используется множество, а на выходе по прежнему желательно иметь один файл. Эту задачу решают т.н. бандлеры.  
  
**Бандлер (bundler)** это инструмент для сборки модулей в единые пакеты, имеющий доступ к файловой системе. Бандлер ищет все выражения require (имеющих ошибочный, с точки зрения браузера синтаксис) и меняет их на настоящее содержимое каждого требуемого файла.

Пример кода, использующий npm-пакет moment.js и корректно работающий в Node.js:

При использовании бандлера мы получим из такого кода единый работающий js-файл без выражений require.

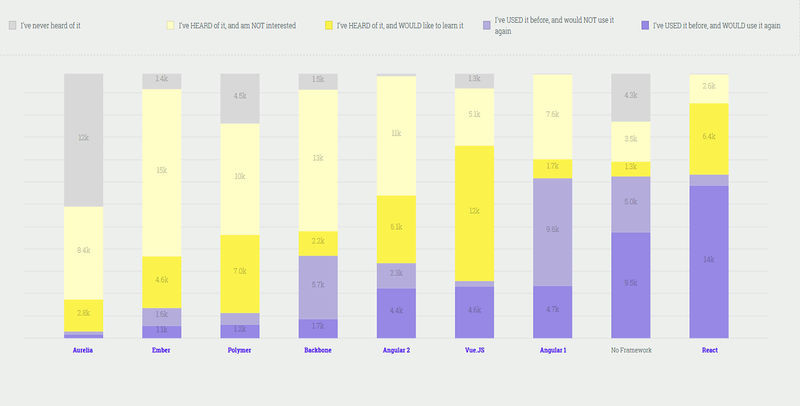
Небольшой оффтоп про стремление делать круто не привлекая лишние сущности

Собственно, теперь можно перейти к обзору популярных фреймворков, каждый из которых сейчас можно загрузить не только в виде отдельного файла, но и с помощью менеджера зависимостей, в виде модуля.

JavaScript-фреймворки

Сегодня самые популярные направления, охватывающие, пожалуй, процентов 80 всей сферы фронтенда это «большая троица»:

* **React** — библиотека, разработанная, в первую очередь, для SPA и реализующая концепцию т.н. «реактивного» взаимодействия между элементами и предоставляющая шаблонизатор JSX, позволяющий описывать компоненты интерфейса с помощью синтаксиса JS. Основная идея, пожалуй, в использовании собственной системы событий, которая и обеспечивает эту самую «реактивность» (тут могу ошибаться, буду рад, если меня поправят). Насколько я могу судить, чаще всего используется в связке с Redux (Redux это одно из решений, реализующее «реактивную» архитектуру Flux). React придумали ребята из Facebook.
* **Angular** — мощнейший фреймворк, предоставляющий огромные возможности для разработки приложений, на нём можно реализовывать как «классические» небольшие SPA, так и действительно огромные проекты. Этот фрейм, в полной мере соответствует концепции MVC, предоставляет возможность использовать компоненты, имеет свою удобную систему событий и вообще, по моему субъективному мнению, всячески хорош. Если вы планируете разрабатывать большой серьёзный проект, я бы рекомендовал изучать именно его. С выходом версии 2.0. проект разделился на два независимых — это AngularJS и просто Angular (нумерация версий начинается с двойки), не следует их путать.
* **Vue** — популярный ныне фрейморк. Я с ним не работал, поэтому могу лишь констатировать то, что понял из чужих статей. Насколько могу судить, развитие данного фреймворка испытывает значительное влияние React — он также как и React использует виртуальный DOM, а вторая его версия поддерживает шаблонизатор JSX. Судя по всему, это попытка упростить чрезмерно сложные фреймворки типа React или Backbone, выкинув лишнее, поскольку, например, для реализации своей системы событий тут необходимо подключать дополнительный модуль (Vuex), а ядро фреймворка используется в первую очередь как шаблонизатор и селектор элементов. Повторюсь, у меня нет опыта работы с этой библиотекой, я могу ошибаться и буду рад, если кто-нибудь меня поправит.

Сравнительная оценка популярности JS-фрейморков в 2018-м, по результатам большого опроса Stateofjs.com (кликабельно):  
  
[](https://habrastorage.org/webt/ed/rx/05/edrx05wopipril_5bf1x7w9adxi.png)  
  
Как видно, на сегодняшний день также считаются популярными **Polymer**, **Ember** и **Backbone** (тут спорно, особенно учитывая, что последний раз его обновили в феврале этого года, а до этого обновления не было года три) и не упомянутый в статистике **Meteor**. Однако, анализируя предложения о работе, можно видеть, что первая троица имеет наибольший спрос, поэтому описывать остальные проекты я не вижу смысла.  
  
Разумеется, библиотек куда больше, и даже такие «древности» (тут сарказм) как jQuery ещё используются повсеместно. Но для понимания современного WEB вполне достаточно быть знакомым с этой троицей. Хотя тот же jQuery для освоения гораздо проще, чем три упомянутых выше библиотеки — при хорошем знании JS в нём разобраться можно буквально за вечер (и если вы этого не сделали, то определённо стоит).  
  
Пара слов о современных тенденциях разработки фронтэнда. Не хотелось бы пускаться в критику использования реактивного подхода для больших проектов, но всё-таки стоит упомянуть о том, что подход SPA для больших приложений это пока ещё не лучший выбор. Например, на том же Facebook мы можем наблюдать как первая же страница (логин) грузит с сервера больше 4 МБ данных, причём это всё только JS-код. И, если верить тому, что пишут о кодовой базе лицокниги, там есть не только React, но и Angular. Забавно, не правда ли? Если ваш фреймворк такой уж хороший, зачем вам ещё и Angular?  
  
Важное отличие между React и Angular — тип связывания данных модели и представления. В Angular есть как двустороннее, так и одностороннее связывание. Поясню на примере — двустороннее связывание это когда элемент пользовательского интерфейса (поле ввода, чебокс, селект) обновляется, Angular автоматически обновляет данные в модели. React же работает несколько по другому — сначала обновляется модель и затем рендрится элемент, а для изменения модели нужно использовать callback-функции. Где-то удобнее первый вариант, где-то второй. Однако для работы с React нужно чётко понимать, что именно происходит в системе событий приложения.  
  
Сейчас я пишу для двух крупных закрытых проектов, в одном из них абсолютно всё реализовано на чистом AngularJS и всего лишь паре сторонних модулей для него. За четыре месяца я не получил ни одной задачи, которую нельзя было бы решить силами этого фреймворка, хотя некоторые вещи выглядят, конечно, не так красиво как это можно было бы реализовать в React'е, но это скорее частные случаи — как пример можно привести концепцию создания индивидуального scope для каждого компонента, это может быть весьма затратно и неудобно, если компонентов на странице под сотню и они имеют сложное взаимодействие.

Современный подход к CSS-вёрстке

На чистом CSS сейчас верстают относительно редко. Причин тому множество, одни из самых очевидных это усложнение стилей пропорционально сложности проекта и несовершенство CSS-селекторов.  
  
Для расширения возможностей CSS сейчас используются SASS и LESS — это динамические языки CSS стилей, которые могут компилироваться как на стороне сервера (SCSS), так и на стороне клиента (LESS, хотя есть и реализации для server-side). Оба этих языка обеспечивают следующие расширения CSS: переменные, вложенные правила, миксины, операторы и даже (ограниченно) функции.  
  
Вопросы конкретной их реализации я упомяну лишь вкратце, это отдельная широкая тема, в каждом проекте это реализуется по разному. Языки эти не особо сложные, для их понимания в большинстве случаев достаточно знания CSS и нескольких правил конкретного языка. Cитуацию дополнительно упрощает тот факт, что любой валидный CSS должен корректно распознаваться этими языками.  
  
Чуть подробнее:

* LESS — это вложенный метаязык: валидный CSS будет валидной LESS-программой с аналогичной семантикой. LESS разработан с целью быть как можно ближе к CSS, поэтому у них идентичный синтаксис. В результате существующие CSS можно использовать в качестве LESS кода.
* SASS — это метаязык на основе CSS, предназначенный для увеличения уровня абстракции CSS кода и упрощения файлов каскадных таблиц стилей.

Последние версии языка Sass имеют два синтаксиса:

* SASS — более краткий, отличается отсутствием фигурных скобок, в нём вложенные элементы реализованы с помощью отступов;
* SCSS (Sassy CSS) — использует фигурные скобки, как и сам CSS.

Пара важных различий между LESS и SASS

Пожалуй, на этом общие тенденции можно считать описанными, и теперь стоит перейти к одной из самых спорных и одновременно популярных технологий в современной web-разработке. Пришлось даже выделить её в отдельный раздел, поскольку она порой размывает границы между бэкэндом и фронтэндом, предоставляя возможность использовать полноценный SSR, используя при этом JS.

Node.js

Node.js уже достаточно зрелая технология, продолжающая развиваться семимильными шагами. Это интерпретатор языка JavaScript для серверной стороны (поначалу звучит бредово, да). Сам по себе Node.js является C++ приложением, которое получает на входе JavaScript-код и выполняет его. При этом, нода имеет встроенный веб-сервер (весьма удобный в использовании для мелких проектов, хочу заметить).  
  
О том, как вообще подобная идея появилась очень хорошо [написано тут](https://habr.com/ru/post/108241/). Статья старая, но для понимания прочесть стоит.  
  
Функциональность Node.js расширяется с помощью пакетов. Пакетом в Node.js называется один или несколько JavaScript-файлов, представляющих собой какую-то библиотеку или фреймворк.   
  
NPM (аббр. node package manager) — это стандартный менеджер пакетов, автоматически устанавливающийся вместе с Node.js. Он используется для скачивания пакетов со своего облачного сервера, а также для загрузки пакетов на эти сервера. Он же отвечает за соблюдение зависимостей между используемыми в проекте библиотеками.  
  
Реалии таковы, что если вы сегодня пишете для фронтэнда в крупном коммерческом проекте — в абсолютном большинстве случаев это необходимость ставить ноду для использования фреймворков. Разумеется, если вам надо сверстать буквально одну простую страницу, отправляющую некие данные на сервер, сделать это, конечно, можно просто подключив один файл подходящего вам фреймворка и используя его базовые функции, однако с нодой никто уже так не делает.  
  
Если вы внимательно прочли предыдущую часть статьи, сейчас вам должно стать понятно, как Node.js объединяет фронт и бэк. Используя ноду вы можете писать код на JS (или на чём вам больше нравится — TypeScript, CoffeeScript, ES6 и т.д.), используя возможности шаблонизатора выбранного фреймворка.  
  
Это, в общих чертах, может выглядеть следующим образом:

1. Клиент передаёт запрос серверу (который реализован на ноде) в виде адреса запрашиваемой страницы.
2. Нода парсит запрос и передаёт его «точке входа» вашего JS-приложения.
3. Приложение анализирует запрашиваемый адрес и формирует нужную страницу из компонентов, описанных с помощью шаблонизатора используемого фреймворка.
4. Перед отдачей результата клиенту, специальные пакеты ноды собирают все используемые JS-библиотеки в один файл, проводят компиляцию SASS/LESS, делают ещё кучу полезных вещей (тестирование компонентов, например) и возвращают клиенту готовый HTML с всего двумя подключаемыми файлами (JS и CSS).

Надеюсь из описания процесса наглядно понятно, что я имею в виду под объединением фронта и бэка.  
  
Однако, очевидно, что если каждый раз выполнять рендеринг компонентов, постпроцессинг и сборку в один файл, это будет слишком долго. Поэтому на практике, именно такой подход используется только при разработке. Для загрузки в «продакшн» запускают билд проекта, получив на выходе уже готовые файлы, которые уже можно отдавать клиенту с помощью вполне традиционного бэкэнда (PHP, Ruby, да хоть сами сервер на C++ напишите, словом, как угодно). Иными словами, для разработки и поддержки проекта ставить ноду вам в любом случае придётся, поскольку править что-то в этих готовых файлах без исходников проекта это то ещё веселье.  
  
Думаю очевидно, что всё это открывает огромный простор для разработки и широчайший выбор архитектур для построения проекта. Не то что стандартный набор (PHP + HTML\CSS + jQuery ) лет десять назад.  
  
А ведь ещё можно, например, ещё больше унифицировать части кода и использовать их при разработке не только web-приложения, но и мобильного приложения, создавая цельную экосистему (Weex, React Native).  
  
Словом, Node.js, это круто, это хорошо. Осталось чуть подробнее описать самые популярные ныне пакеты и их назначение.

Некоторые популярные пакеты Node.js для разработчиков

* **Babel** — транспайлер кода разных стандартов в валидный JS для разных «рантаймов» (старые браузеры и т.п.). Не содержит в себе правил преобразования, они идут отдельными пакетами. Без этого инструмента сейчас не обходится практически ни один проект.
* **Grunt**— менеджер задач для автоматического выполнения рутинных операций (например, минификация, сжатия изображений, тестирование, объединение файлов). Использует командную строку для запуска задач, определённых в файле Gruntfile.
* Примеры бандлеров для Node.JS: **Webpack** и **Browserify**. С помощью Webpack и встроенного NPM, например, можно легко заменить связку Bower и Gulp/Grunt.
* **Yarn** — по сути, аналог NPM, оптимизирующий некоторые аспекты, например загрузку и установку пакетов. Использует тот же формат package.json и те же типы пакетов, что NPM.
* **Bower** — менеджер пакетов предназначенный в первую очередь для подключения JS и CSS библиотек и фреймворков. По сути, аналог NPM, но только для фронта. Раньше был популярнее, сейчас считается устаревшим и, что важнее, небезопасным — даже сами авторы Bower призывают мигрировать на более современные инструменты, типа связки Yarn и Webpack.
* **Jade** — по сути это шаблонизатор HTML, написанный специально для ноды, имеющий также некоторую функциональность JS. На мой субъективный взгляд идея в целом неплохая, но мне не нравится его синтаксис. Правда есть и ещё один сомнительный момент — если проект небольшой, спорна необходимость использовать шаблонизатор. А если большой, то один из трёх популярных JS-фреймворков скорее всего будет использован и возможности любого из них, по факту, функциональность Jade полностью перекроют.

\*\*\*  
  
Чуть-чуть новомодного сленга, на закуску:  
  
**Boilerplate code** или просто boilerplate — под этим подразумеваются секции кода, которые должны быть написаны во многих местах с минимальными изменениями или вообще без них — например, код шапки HTML-страниц в шаблонах. Также может использоваться по отношению к языкам, в которых программист должен написать много кода, чтобы выполнить минимальную задачу — например объявление класса с приватными переменными и написание внутри публичных функций, только возвращающих значение этих приватных переменных.  
  
**DevOps**(акроним от англ. development и operations) — набор практик, нацеленных на активное взаимодействие специалистов по разработке с сисадминами и взаимную интеграцию их рабочих процессов друг в друга. Проще говоря, любому нормальному fullstack-разработчику без девопса никуда, ведь для того чтобы самостоятельно развернуть себе нормальную среду для разработки, надо хотя бы в общих чертах понимать как работает веб-сервер и как его настроить.  
  
Пожалуй, на этом краткий обзор современной web-разработки можно завершить, тем более что он и так получился не таким уж и кратким. Буду рад любым замечаниям.  
  
\*\*\*  
  
Upd. В качестве бонуса. Спасибо [KonstantinSpb](https://habr.com/ru/users/konstantinspb/).

[Stackshare.io](http://stackshare.io/) — чтобы быть в курсе, что используется в топовых и не очень компаниях.  
[Libhunt.com](http://libhunt.com/) — отличный категорированный список по огромному числу технологий со сравнительной статистикой.