Хочу стать frontend разработчиком: базовые знания и план обучения

Итак, друзья, вы решили встать на путь программирования? Жму руку, это решение изменит вашу жизнь. Это всегда интересная и востребованная работа на стыке интеллекта и творчества, работа о том, как принести пользу людям, сделать мир чуточку лучше. Возможности реализации в ней бесконечны.

В программировании много разных областей: веб-разработка, мобильная, десктопные приложения, разработка ОС, драйверов для железа. Веб-разработка — одна из самых интересных и востребованных областей. К её плюсам можно отнести то, что ваш продукт лежит в Интернете, и чтобы его увидеть, достаточно набрать адрес в браузере любого устройства, не нужно ничего качать и устанавливать. К тому же, с помощью современных инструментов, зная веб, можно разрабатывать сразу и мобильные, и десктопные приложения. Веб состоит из frontend (то, что видит клиент в браузере) и backend (серверная часть, занимается хранением, обработкой и выдачей данных). Я предлагаю начать знакомство с вебом именно с фронтенда.

Да, кстати, меня зовут Роман Латкин, я почти 10 лет варюсь в веб-разработке. Когда я начинал, всё было одновременно просто и сложно. Просто, потому что для построения приложения много знать было не нужно: вот HTML, немного CSS, чуть-чуть JavaScript — и готово. Сложно, потому что разработка велась через боль. Сейчас множество этой боли вылечено с помощью громадной экосистемы инструментов, но она очень пугает новичков, они не знают, как подступиться к фронтенду, с какой стороны подойти. Мне повезло, я наблюдал развитие фронтенда почти с начала, и у меня в голове всё неплохо уложилось. И я хочу в помощь начинающим разработчикам передать это понимание. Надеюсь, после прочтения этой статьи, вы будете чётко знать, каким путём идти, куда копать и по какому плану развиваться.

Три составляющих фронтенда

Весь фронтенд состоит из трёх составляющих: HTML (содержание и разметка), JavaScript (логика) и CSS (внешний вид, позиционирование). HTML описывает содержание страницы и выглядит примерно так: `<table></table>`. CSS описывает стили и выглядит вот так: `table { background: #ccc; }`. JavaScript — язык программирования, описывает логику приложения, а также обращается к элементам HTML, изменяя структуру и содержание страницы (пример кода: `var count = 5; count = count + 5; console.log(count) // 10`).

Если вам придётся как-нибудь вручную использовать эти средства, чтобы сделать более-менее сложный проект, то вам предстоит испытать много боли. К счастью, мир развивается, сложные вещи упрощаются, люди придумывают всё новые и новые инструменты и технологии, которые делают этот созидательный процесс более приятным и быстрым.

Любой процесс познания можно представить в виде буквы «Т», где горизонтальная линия — широкое понимание, вертикальная — глубокое. У идеального специалиста буква Т большая и красивая, равномерная. Если она вытянута в одну сторону, она некрасива, уродлива; такой специалист мало полезен в боевых делах. Он может либо глубоко разбираться в чём-то одном, но чуть шаг в сторону, и он непригоден; либо поверхностно разбираться во всём, но при этом ничего не уметь. В первую очередь необходимо максимально развить широкую составляющую, чем мы сейчас и займёмся — постараемся максимально широко охватить все аспекты фронтенда, не углубляясь. А потом вы займётесь углублением, которое останется вам на самостоятельную работу.

Чтобы хорошо представлять причины текущего положения вещей, нужно хотя бы чуть-чуть знать историю пути развития фронтенда, да и вообще веб-приложений в целом. Инструментов сейчас столько, что не только новичок, сам чёрт ногу сломит. Поэтому, чтобы в этом всём хорошо ориентироваться, начнём именно с истории, с короткой экскурсии, как всё начиналось, а затем плавно перейдём к современным подходам.

Первые сайты

Вначале люди писали на чистом HTML, рисовали внешний вид на чистом CSS, делали логику на чистом JavaScript. Типичное старомодное приложение — это когда серверная логика генерирует HTML (отвечая на запрос посетителя, сервер берёт данные из базы данных и вставляет их в HTML) и отдаёт его вместе со статическими файлами стилей и клиентской логики на JavaScript, которой в то время (около 10 лет назад) было немного. При совершении перехода на другую страницу весь этот процесс повторялся. То есть раньше как такового разделения на фронтенд и бэкенд не было, было одно цельное приложение, которое одновременно и работало с базой данных, и генерировало HTML.

jQuery

Писать на чистом JavaScript надоедало и появился хороший инструмент — jQuery, который с помощью удобного синтаксиса позволял обращаться к элементам страницы и выполнять с ними какие-то действия. Появлялись различные плагины, готовые решения, стало проще и интереснее.

> Чтение по теме: Краткая шпаргалка по jQuery

Но приложения развивались, объём клиентской логики рос, и постепенно всё это превращалось в большую лапшу. Чтобы её распутать, нужна была какая-то форма, архитектура.

MVC

Умные Парни попробовали перенести на фронтенд архитектурный шаблон с серверной части — MVC (модель-представление-контроллер). Этот шаблон диктует правило, что есть модель, которая описывает данные. Например, модель пользователя, модель фильма, модель отзыва. Есть контроллер, который обрабатывает запросы, например «показать по такому-то адресу страницу со списком фильмов». И есть представление, которое отвечает за отображение данных в HTML, в которое контроллер передаёт готовые данные, полученные из базы данных/API.

Здесь началась история single page application, SPA — приложений, которые загружаются один раз, а затем при переходе по страницам обращаются к серверу за данными по API. Этот подход называется AJAX. Вместо того, чтобы генерировать HTML на стороне сервера, сервер отдаёт клиентскую логику приложения один раз. Переходя на другую страницу, например с главной страницы на страницу поиска отелей, приложение запрашивает с сервера данные в чистом виде (к примеру, информацию об отелях), без тегов HTML (как правило в формате JSON), и самостоятельно генерирует представление.

Шаблон MVC на фронтенде был хорош, прекрасно работал, но было излишне сложно. Angular, Backbone — представители этой вехи истории. Они, к слову, живут и сейчас, но я в них глубоко не разбирался.

Процессоры и сборщики

Приложения начали расти в размере, и тут пришло время рассказать о сборщиках, препроцессорах и пакетных менеджерах. Постараюсь вкратце по ним пробежаться, несмотря на то, что они заслуживают отдельной статьи.

В вебе важна скорость, поэтому нельзя просто так отдавать посетителю большие файлы, они будут идти по сети слишком долго. Поэтому все ресурсы сжимаются с помощью разных минификаторов. JavaScript чаще всего с помощью uglify (он удаляет пробелы, делает названия переменных короче и ещё много чего интересного). В CSS удаляются пробелы и могут ещё объединяться некоторые свойства. И всё это собирается в один или несколько файлов вместо 10-20, один файл скачать гораздо быстрее, и на сервер нагрузка меньше.

CSS

Что касается CSS, появлялись так называемые препроцессоры. Они расширяют синтаксис CSS, добавляют туда кучу разных возможностей — вложенные блоки, переменные, циклы. Даже просто отсутствие точек с запятой очень помогает и ускоряет написание кода?.

Препроцессор — это такая программа, которая запускается и компилирует этот сахарный синтаксис в чистый CSS. Использование препроцессоров позволяет избежать повторного использования кода, выстраивает архитектуру, и по сути превращает язык описания стилей в язык программирования. Изучите какой-либо инструмент, и вы поймете. Я для себя сейчас выбрал Stylus; есть ещё несколько, например — LESS, SASS.

Чуть позже придумали постпроцессоры. Они, в отличие от препроцессоров, обрабатывают уже готовый CSS, модифицируя его: например, добавляя дополнительные свойства к уже существующим, или изменяя названия классов, делая их уникальными, чтобы ничего не сломалось. Их чаще всего используют для поддержки кроссбраузерности, о которой мы ещё поговорим ниже. PostCSS— вот этот самый постпроцессор, он ещё обладает большой библиотекой плагинов, упрощающих жизнь.

JavaScript

Насчёт JavaScript: исторически так сложилось, что этот язык изначально был слишком простой и сейчас постоянно развивается, обрастая новыми инструментами. Основная его версия, которая работает во всех современных браузерах, называется ES5. В 2015-м году появился усовершенствованный стандарт JavaScript ES2015, или ES6, который даёт много новых инструментов упрощённого описания логики. Только он не работает в старых браузерах, поэтому используют препроцессор Babel для компиляции его в ES5. То есть код пишется с помощью современного синтаксиса ES6, а для работы в браузере сразу компилируется в ES5.

Есть ещё разные способы писать нормальный код, которые сводятся к тому же: код пишется на своём «особом» языке (как в случае с ES6), а потом транслируется в JavaScript. Вот некоторые из этих «особых» языков программирования:

- TypeScript — он добавляет к JavaScript множество инструментов из серьёзного программирования — классы, интерфейсы, модули и др., а также упорядочивает типы переменных. Он больше для того, чтобы писать массивную логику, пользуясь приёмами строгой типизации, и подходит скорее для отдельных крупных логических модулей;
- CoffeeScript — делает код намного более удобным, понятным, человечным;
- и ещё много разных — Dart, Elm, я их глубоко не изучал.

HTML

Для упрощения написания HTML, чтобы не ломать пальцы о теги, стали использовать препроцессоры HTML. Они позволяют, например, вместо громоздкой конструкции `<a href="#">Ссылка</a>` с кучей угловых скобочек писать просто `a(href="#") Ссылка`, а потом компилировать это всё в HTML. Очень рекомендую сразу же освоить Pug, сокращающий объем написанного практически вдвое.

Чтобы удобно вставлять динамические данные в HTML, отделяя данные от разметки, придумали шаблонизаторы. Например, в разметке пишется `<h1>{{ title }}</h1>`, запускается шаблонизатор со значением переменной `title`, и это значение подставляется вместо фигурных скобочек. Теги отдельно, контент отдельно. Можно удобно вставлять динамический контент с помощью циклов и условий — например, передавать массивы объектов и выводить их в таблице.

Менеджеры пакетов

Чтобы не изобретать велосипеды, разработчики давно научились делиться между собой готовыми участками кода, модулями. Во фронтенде для этого активно используется менеджер зависимостей npm. На npmjs.com можно найти огромное количество модулей, плагинов, библиотек на все случаи жизни. Прежде чем писать что-то своё, поищите там.

Позже появился усовершенствованный менеджер зависимостей Yarn, он делает всё быстрее и стабильнее, не буду сейчас углубляться, почему.

Менеджеры задач

Для того, чтобы централизованно управлять всем этим зоопарком, появлялись менеджеры задач. Они позволяют в одном месте описать все процессы и этапы сборки приложения. Это Grunt, Gulp, Webpack. Последний — наиболее подходящий для сборки веб-приложения. Он может взять на себя много забот, легко и просто компилировать все ресурсы, будь то скрипты, стили, разметка, картинки — в любом формате (Stylus, Less, Sass, ES6, TypeScript, jpg, png) из любых исходников — в единые бандлы, сборки файлов js, CSS, HTML, которые будут работать в браузере.

> Чтение по теме: Webpack — основы настройки проекта на JavaScript и Sass

Компонентная архитектура

Итак, можно продолжить: сложное начали упрощать, и в ходе упрощения, большого упрощения, Умные Парни решили — всё есть компонент. Кнопка — это компонент, шапка — компонент, выбор города — компонент. Страница — тоже компонент. Компонент может содержать в себе другие компоненты. Получилось крайне просто: такая концепция, как оказалось, пришлась как родная к

построению клиентских приложений.

Что такое компонент? Это самостоятельный и независимый участок разметки со своей логикой и стилями. У компонента есть свое текущее состояние. Открыто ли меню, активна ли вкладка, и т.п. Состояние всего приложения можно представить как дерево состояний различных компонентов.

Разметка HTML зависит от текущего состояния, изменилось состояние — изменилась разметка. Это реализуется с помощью технологии Virtual Dom — когда DOM (дерево HTML-элементов страницы) рассчитывается сначала виртуально и в конце расчёта отображается в реальном DOM, в разметке. За счёт этой идеи достигли более высокой производительности приложений, ведь одна из самых тяжёлых частей работы браузера — операции с DOM (работа с деревом объектов HTML).

Здесь важно ввести ещё одно понятие — реактивные приложения. Это, упрощённо говоря, когда вместо прямого изменения DOM/Virtual Dom при изменении данных, вводится объект состояния, модель данных, и на её изменения подписывается обработчик, который уже меняет DOM. То есть чтобы что-то поменять в представлении, HTML (например, таблица со списком пользователей), нам достаточно изменить свойство модели (добавить в массив нового пользователя), всё остальное произойдет само (пользователь появится в html-таблице). Вы, наверное, замечали, что некоторые сайты медленно работают, а другие молниеносны. Скорее всего, первый на jQuery и работает с реальным DOM, второй — на одном из реактивных инструментов, с которыми мы познакомимся далее.

React

Итак, эти концепции (Virtual Dom, компоненты, реактивность) улеглись в новом инструменте создания клиентских приложений от Facebook — React. На текущий момент он является одним из лидеров индустрии, наиболее часто используемым во фронтенде. Он обладает развитой экосистемой — можно найти огромное количество готовых компонентов и дополнений.

Управление состоянием

Но между компонентами нужно было наладить связь, им нужно общаться между собой. Нажали на кнопку — изменился цвет. Можно строить эту взаимосвязь напрямую, но это быстро может превратиться в кашу. Тут придумали шаблон централизованного управления состоянием, когда есть одно место, где хранится состояние всего приложения в текущий момент времени. Это, сильно упрощая, такой JavaScript-объект со свойствами. Это состояние изменяется с помощью вызова действий и мутаций, но не будем сейчас так углубляться. Паттерн называется Flux. Самая популярная имплементация управления состоянием для React — Redux.

React, хоть он и обрёл большую популярность и развитую экосистему, на практике оказался слишком голый, сложный, многословный. Для того, чтобы сделать простую форму, нужно много-много лишнего кода написать и кучу дополнений поставить. Для того, чтобы создать простое приложение, нужно ещё много чего установить, и вариантов для простых вещей очень много, легко можно потеряться — экосистема хоть и огромная, но в ней сложно ориентироваться; внешне одинаковые приложения могут быть совсем по-разному устроены внутри. А концепция JSX — переплетение кода и разметки, выглядит не столь удачно, сложно потом понимать, что же такое хотел сказать автор кода, сложно менять разметку.

Vue.js

Тут появился Vue.js — гибкий, эффективный и простой в освоении веб-фреймворк, который несёт в себе всё те же концепции, но они в нём выглядят гораздо удачнее. Он объединил в себе всё лучшее из Angular и React, более чётко ответил на вопрос «что есть что». Из коробки Vue содержит уже большое количество инструментов и возможностей, которые в несколько строк позволяют писать объёмную логику. Разработка значительно упростилась.

Vue принёс ещё несколько интересных концепций, как, например, однофайловые компоненты — файлы, которые содержат в себе сразу логику, разметку и стили, и они там не переплетаются, как в случае с React и JSX. Vue из коробки позволяет использовать любые препроцессоры, которые очень органично вписываются в однофайловые компоненты. И имеет множество готовых встроенных решений, даже свою имплементацию Flux. Vue обладает отличной документацией на русском языке, которая научит вас лучшей практике во фронтенде, от сборки приложения до автотестов.

Изоморфные приложения, SSR

В разговоре об одностраничных приложениях мы упустили одну важную деталь: когда поисковый робот обращается к одностраничному приложению, он ничего не видит — только пустую страницу с тегами body без контента. В старомодных приложениях сервер обратился бы к базе данных, сгенерировал представление и отдал бы готовый HTML с текстом страницы. В случае с одностраничным приложением сервер отдаёт пустую страницу, которая лишь после инициализации подтягивает данные и показывает представление, чего конечно же поисковый робот не сделает. Таким образом, использовать одностраничные приложения для сайтов, ориентированных на контент, SEO, недопустимо.

Это недопущение обходилось множеством хаков и костылей, пока не появилась концепция SSR — Server-Side Rendering. Умные Парни научили весь JavaScript, который работал в браузере, выполняться на сервере с помощью NodeJS(технология создания серверных приложений с помощью браузерного языка JavaScript). Это, конечно, ввело свои ограничения, но жить стало легче. Теперь можно было написать логику один раз на одном языке, и она сразу же работала и на сервере (при первом обращении посетителя/робота генерировался HTML с контентом страницы) и в браузере (последующие переходы посетителя). Это и называется изоморфное, универсальное приложение.

Схема простая: при первом заходе посетитель отправляет запрос на сервер NodeJS, который обращается к API-серверу, берёт данные в виде JSON и отрисовывает их в HTML, возвращая посетителю. Дальше уже приложение живёт в браузере, при последующих переходах по страницам оно напрямую обращается к API-серверу за данными и уже непосредственно в браузере отрисовывает представление.

В React имплементация этой схемы делается разными и сложными путями. В качестве готовых решений есть для этого, например, фреймворк Next.js. В документации Vue есть целый раздел, посвященный SSR. Там указан фреймворк Nuxt — Vue + SSR. С его помощью можно довольно легко писать такие универсальные приложения.

CSS-фреймворки, адаптивность

Теперь мы сменим тему на попроще и поговорим о вёрстке.

Исторически, чтобы создать сетку страницы, её каркас, в первые времена верстальщики использовали таблицы. Потом начали использовать блоки, или контейнеры, появилась контейнерная вёрстка. Положение блоков устанавливалось с помощью свойства позиционирования `float: right/left`.

В настоящее время всё упрощается, уже почти все браузеры поддерживают Flexbox и CSS Grid — современные удобные способы верстать сетку страницы. Их умелое сочетание позволяет в несколько свойств добиваться таких положений контейнеров, над которыми пришлось бы старыми методами изрядно попотеть, позволяя с лёгкостью выполнить практически любые дизайнерские изыски.

> Чтение по теме: Flexbox и Grid — знакомство с CSS-вёрсткой

Адаптивность — это способность страницы выглядеть одинаково хорошо на всех устройствах, будь то ноутбук, планшет или мобильный телефон. Адаптивность достигается с помощью медиа-запросов — блоков условий в CSS, при каких разрешениях экрана какие CSS-свойства должны работать. Её можно также добиться с помощью умелого применения flexbox-контейнеров.

Все веб-приложения в основном типичны, состоят из строк, колонок, таблиц, кнопок и других UI-элементов. Чтобы не писать их каждый раз, в помощь сайтостроителям создавались CSS-фреймворки, где вся разметка уже продумана — достаточно применить нужный класс. Они содержат в себе множество готовых UI-элементов. Самый популярный — конечно же Bootstrap, сейчас уже 4-я версия. Есть ещё Bulma, тоже довольно хороший. И ещё множество менее популярных. Обычно в CSS-фреймворках адаптивность идёт из коробки, важно лишь правильно пользоваться предлагаемыми инструментами. CSS-фреймворки станут отличной основой практически в любом вашем веб-приложении и хорошим началом освоения навыков правильной вёрстки. Их стоит использовать, когда нужны типичные элементы пользовательского интерфейса, адаптивность, а это 99% кейсов в вебе.

Кроссбраузерность

Это слово означает способность сайта отображаться одинаково в разных браузерах. Как правило, CSS-фреймворки берут эту заботу на себя, но я вкратце расскажу, как это достигается. Для начала нужно обнулить все свойства стандартных элементов (разные браузеры отображают стандартные элементы — списки, таблицы и др. по-разному). В CSS-фреймворках для этого часто можно увидеть специальный файлик — reset.css. Следующее — исторически так сложилось, что браузеры развивались по-разному, и теперь некоторые CSS-свойства нужно прописывать специально для каждого браузера, используя префиксы — `-webkit`, `-moz`. Эту работу можно делать автоматически с помощью вышеупомянутого PostCSS и его autoprefixer.

Методологии

Чтобы вёрстка не превратилась в суп, ничего внезапно не ехало, всё было чётко и красиво — существуют специальные подходы, сборники правил о том, как называть тот или иной класс. Они очень вписываются в компонентную архитектуру, надо сказать, с них она и началась. Правило то же — всё есть компонент, или по-другому «блок». У блока есть свои элементы, мини-блоки, из которых и состоит блок. Изменяют отображение блока модификаторы, применяя к нему то или иное свойство. Изучите БЭМот Яндекса или SUIT CSS, прежде чем начинать заниматься верстанием.

В путь!

Надеюсь, к концу статьи у вас уже сложилось более-менее полное и широкое понимание всех аспектов фронтенда. Теперь вам остаётся лишь его углублять, следуя шаг за шагом. Предложу вам план этих шагов, как стать профессиональным фронтендером:

1. Изучите основы вёрстки — HTML, CSS. Хватит только основ — остальное наработается в процессе решения задач. Сразу для работы поставьте себе редактор VS Code. Отдельное внимание уделите навыкам работы с Flexbox и CSS grid.

2. Изучите Bootstrap или bulma.io. Попробуйте создать каркас простого сайта с их помощью; изучите их исходники, они дадут вам хорошее понимание правильной архитектуры проекта. Примерно уже здесь, а лучше как можно раньше, пробуйте собирать какие-нибудь проектики, решать какие-нибудь задачки, нарабатывайте практику.
3. Изучите JavaScript. Да, тут тоже хватит только основ. Пробегитесь по синтаксису ES6, чтобы примерно его понимать. Попробуйте разобрать, как реализованы те или иные UI-компоненты в вышеупомянутых CSS-фреймворках.
4. Изучите основы Git. Это система контроля версий, и она уже на данном этапе хорошо вам послужит, позволит фиксировать поэтапно изменения в коде и хранить их.
5. Изучите BEM/SuitCSS, что больше понравится.
6. Поймите синтаксис Stylus и Pug.
7. Начните изучать документацию к Vue.js. Она предельно понятна и на русском языке. В процессе изучения вы узнаете множество смежных вещей — компонентная архитектура, сборка с помощью webpack, работа с API, SSR, flux, автотестирование.
8. Пробегитесь по библиотеке lodash — она вам очень поможет при написании кода на JavaScript, для более лаконичного кода без велосипедов.
9. Изучите автотестирование фронтенда. Это важный пункт, если вы сразу его освоите, облегчите себе дальнейшую жизнь. Не откладывайте его на потом. Рекомендую такие инструменты, как Jest и TestCafe. В Vue.js есть хороший инструментарий для автотестов из коробки.
10. Создайте собственное приложение, используя полученные знания. Придумайте идею или возьмите ту, что у вас давно сидит в голове; не просто так вы ведь решили стать программистом! В дополнение изучите транслируемые в JavaScript языки — TypeScript, CoffeeScript.

Готово! Дальше только практика, вернее, она должна была начаться с первого пункта, а сейчас достигнуть своего апогея. Теперь вы мастер фронтенда! Хотя кто знает, может, к тому времени опять выйдет в свет какой-нибудь инструмент, который всё перевернёт во фронтенде, и придётся полностью менять свои понимания?

Глубоко в каждой теме не закапывайтесь, не старайтесь всё сразу запомнить. Главное — помнить, где и что посмотреть. Никогда не будет лишним повторить основы. Полезно общаться в комьюнити и желательно иметь живого, пусть даже удалённого, наставника, который поможет направить в случае застоя. Помните, что лучшее понимание приходит в процессе решения задач.

Источник: [tproger.ru](https://tproger.ru/curriculum/intro-to-frontend-development/)

2019-02-15