

Я верстальщик

Веб-верстальщик

Арсений Матыцин

12+

Арсений Матыцин
Я верстальщик. Веб-верстальщик

Введение

Для кого

Данный материал предназначен в первую очередь для тех, кто только начал осваивать веб, не знает за какие ниточки потянуть и что сделать, чтобы понять, как это все работает. Не менее полезен материал будет и дизайнерам, которые рисуют макеты и слабо представляют, что с этим потом делает верстальщик.

Скорее, он даже очень нужен, так как давно существует проблема взаимопонимания между дизайнерами и верстальщиками.

В целом данный материал создавался для повышения компьютерной грамотности среди людей хоть как-то связанных с версткой.

Что такое верстка

Существует несколько видов верстки. Изначально этот термин использовался для обозначения монтажа текста, изображений на макете, который затем переносился на бумагу, и впоследствии, с приходом интернета, версткой стали называть процесс создания гипертекстовой разметки (HTML кода).

Верстка сайта предполагает перенос дизайна макета в формат удобоваримый для браузера. Задача верстки сегодня – создать сайт, который одинаково хорошо будет смотреться на всех устройствах.

Будем откровенны, задача не из легких и, порой, в принципе неосуществима. Все дело в том, что устройства разные, разработчики железа постоянно что-то меняют, создают новое [иначе бы не было прогресса], но верстальщику из-за этого приходится подстраиваться. Не считая того, что технологии и стандарты верстки тоже постоянно дополняются и меняются. О концепциях и подходах поговорим позже.

В чем отличие от типографской верстки

Когда ты работаешь со статическим, аналоговым носителем, таким как бумага, ты знаешь наверняка, как будет выглядеть верстка в конечном счете.

Если ты берешь визитку, то необходимо просто определиться с ее форматом, настроить макет, сверстать и напечатать. На этом процесс можно закончить и переходить к следующему макету.

Как я указал выше — в вебе никогда не можешь быть уверен наверняка, с какого устройства пользователь посмотрит на плоды твоей работы.

Кто такой верстальщик

Термин верстальщик идет из типографии, где представитель профессии набирал полосы в соответствии с созданным макетом. В принципе, в плане веб, ничего не изменилось, просто поменялись инструменты, и верстальщик стал работать по сути только с текстом. Теперь же, вместо бумаги результат работы видят в браузере.

Также верстальщика можно называть более конкретно – верстальщик веб-страниц. Не так давно появился термин пришедший извне – frontend-разработчик.

Могу сказать, это мое личное мнение, но то, что должен уметь фронтендщик и якобы хорошо знать верстальщик, все это не серьезно. Просто front-end хорошо сочетается с back-end и имеет оттенок ориентированности не только на веб, но и на приложения.

Основа профессии верстальщика/фронтендщика – грамотность, знание основ, их правильное использование в разных проектах, от верстки одностраничника – до разработки крупного приложения. А также возможность изучать новые технологии для достижения и преодоления новых высот и сложностей.

Стандарты

Стандарты

Ранее создавалось множество версий HTML. Среди них были XHTML, строгий и переходной режим HTML4. Для обозначения используемого стандарта используется доктайп. Следующим образом:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
```

Включен строгий режим стандарта HTML 4.01

Сегодня для верстки сайтов актуален только один формат-стандарт – HTML5. И отлично, что к нему пришли. Самым первым его преимуществом является формат записи стандарта в коде:

```
<!DOCTYPE html>
```

Включен режим HTML 5.x

Ко второму преимуществу можно отнести семантику. Верстать можно и просто с использованием div-ов. Визуально никакой разницы, вот только в верстке важно не только как это выглядит. Очень важно, чтобы ваш код был семантически корректно сверстан. Это облегчит будущее существование сайта на просторах интернета.

Впрочем, если вы верстаете какое-нибудь веб-приложение, удел которого быть закрытым продуктом, то семантической версткой можно пренебречь.

Почему стоит следовать стандартам

Как я уже указывал ранее, задача верстальщика – обеспечить оптимальное отображение информации на всех устройствах. Стандарты, создаваемые и опекаемые организацией W3C, призваны служить именно этой цели.

Почему я об этом говорю? Потому что можно заставить браузер отображать верстку и без объявления доктайпа. Даже можно не указывать тег `html`. И большинство устройств спокойно съедят такой код. Но я очень не рекомендую так делать.

Маленький совет

Этим правилом, писать по стандартам, можно пренебречь, если вам необходимо, скажем, отдельно от всего сделать верстку таблицы, которую в дальнейшем вам необходимо вставить в код.

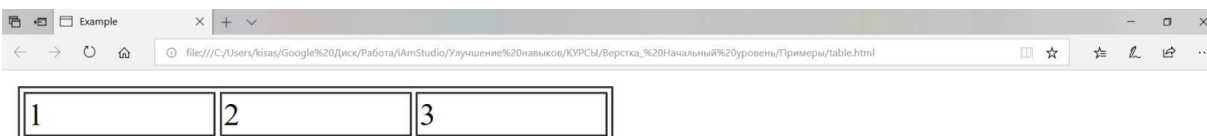
Табличная верстка

Это сильно устаревшая техника верстки, которую я застал, когда обучался дизайну. Тем не менее нельзя обойти стороной то, из чего выросла технология.

Раньше, чтобы расположить элементы в строку рядом друг-с-другом применялись таблицы. Вот пример:

```
<!DOCTYPE html>
<html lang="ru">
  <head>
    <meta charset="UTF-8">
    <title>Example</title>
  </head>
  <body>
    <table width="300" border>
      <tr>
        <td>1</td>
        <td>2</td>
        <td>3</td>
      </tr>
    </table>
  </body>
</html>
```

Пример кода табличной верстки



Результат обработки кода браузером Edge

Сложность такого метода заключалась в том, что невозможно было обойтись без пространственного понимания сетки. Если вы создавали сетку 5 на 5, то вынуждены были следовать ее логике, чтобы разместить информацию внутри. Или создать еще одну таблицу внутри. Принципы табличной верстки хорошо демонстрируются на примере обычной таблицы в редакторе.

Тут в строке `tr` вложить элемент столбца `td`, который занимает место 5 ячеек.

Эта часть таблицы объединяет в себе 2 ячейки по горизонтали и 4 ячейки по вертикали

А справа →,

в ячейку вложена таблица 5 на 5, которая по умолчанию отбивается от границ ячейки отступами.

Если возникала необходимость добавить одну ячейку во второй строке, то она выбивалась из ряда. Чтобы этого избежать приходилось либо в каком-нибудь столбце указывать объединение по вертикали на одну ячейку больше, либо переделывать сетку на 6 в ширину, или же создавать вложенную таблицу.

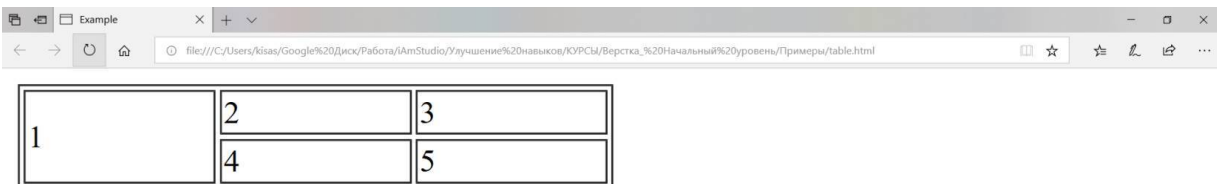
Такие манипуляции по склейке ячеек производятся с помощью атрибутов:

colspan – объединение ячеек по ширине

rowspan – объединение ячеек по высоте

Пример:

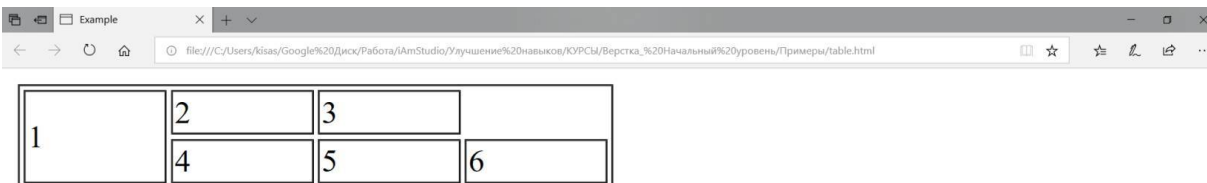
```
<table width="300" border="1">  
  <tr>  
    <td rowspan="2">1</td>  
    <td>2</td>  
    <td>3</td>  
  </tr>  
  <tr>  
    <td>4</td>  
    <td>5</td>  
  </tr>  
</table>
```



1	2	3
	4	5

Результат обработки в браузере

Формально это таблица 3 на 2, но так как я указал первой ячейке расшириться на 2, то было бы ошибкой во второй строке написать такое же количество ячеек `td`, как и в первой. Результат был бы тогда такой:



1	2	3	
4	5	6	

Результат ошибочного кода в браузере

Эти атрибуты легко запомнить, если разбить их на составляющие и знать перевод. Например `span`, который есть в обоих словах – это пролет, объединение, диапазон. В то же время `row` – это строка, а `col` (`column`) – это колонка.

Да и в целом это неплохая практика понимать, что ты пишешь. Необходимо знать английский хотя бы на уровне терминов и тегов, или навыков использования переводчика.

В целом, если вспомнить принцип дизайнера и верстальщика, то это можно воспринимать не как ошибку, а как фичу.

С приходом адаптивного дизайна и верстки над существованием таблиц навис большой вопрос. Так как все попытки отразить сколько-нибудь большую таблицу на экране мобильного телефона оказалось не так просто.

Исторический пример

Таблицы активно использовались в частности для таких вещей, как склейка изображений, создание навигации.

Склейка изображений это вообще обалденная штука, в свое время я увлекался таким сбором, разбором, но лучше всего был фактор оптимизации кода. Изображение, допустим, тигра можно было нарезать и во всех ячейках кроме глаза использовать сжатый формат jpg. А в случае с глазом использования gif-анимацию. Можно сделать gif высокого, насколько это возможно, качества и как результат смотреться будет эффектно и весить не так много, если бы весь тигр был gifкой. (Вес изображения при этом сильно зависит от дизайнера)

Впрочем казалось бы, что такого в табличной верстке и на кой оно вам надо. Ответ прост – она не умерла, а перешла в иное качественное состояние. Сегодня табличная верстка активно используется для создания html-рассылок. В данном случае, в отличие от разработчиков браузеров, разработчики софта для работы с письмами так и не пришли к единому консенсусу и 100% у вас на каждом отдельном устройстве откроются совершенно разные виды на вашу верстку. Порой ужасные.

Чтобы этого избежать там используется не стандарт HTML5, а строгий старый режим HTML4. К нему, вдобавок, идут набором ныне морально устаревшие теги типа center, и приходится использовать ухищрения, которые ни один верстальщик сайтов никогда в здравом уме использовать не будет.

Блочная верстка

Блочная верстка

В отличие от табличной, блочная верстка позволяет набрать необходимое количество элементов страницы без условностей и требований первого метода. Да и в целом этот тип можно отнести к инновационным. Наряду с блочной версткой стали активно использовать внешние файлы css, которые призваны были описывать внешний вид div-ов. Притом эта тенденция (выносить данные) дошла до того, что инлайновые стили стали неким моветоном. Лично я с этим конвейерным и неразумным подходом не согласен.

Разумно использовать внешние стили для повторяющихся элементов, для классов, для медиа запросов и во многих других случаях. Но если в конкретно этом span надо прописать *color:red*, зачем создавать отдельный класс или id и писать под него стили!?

Float:left

Самое частое свойство, которое мне приходилось использовать во время верстки сайта по блочному методу. Так как по-умолчанию `div` занимает всю область по ширине, куда он вписан, то, чтобы выстроить в линию несколько блоков необходимо дать им размер – ширину. Но тогда они просто выстроятся один под одним с указанными размерами. Чтобы выполнить «обтекание» нужно использовать свойство *float*.

Float:left заставляет элемент отобразиться таким образом, чтобы предыдущий был по левую сторону от него.

Небольшой хак

Существует иной способ решить этот вопрос, переноса блоков. Достаточно сменить ему свойство *display* с дефолтного *block* на *inline-block*. Тогда браузер будет воспринимать данный *div*, как символ в предложении.

Данная тема постепенно приводит нас к каскадным стилям. О них я расскажу после того, как опишу работу с тегами.

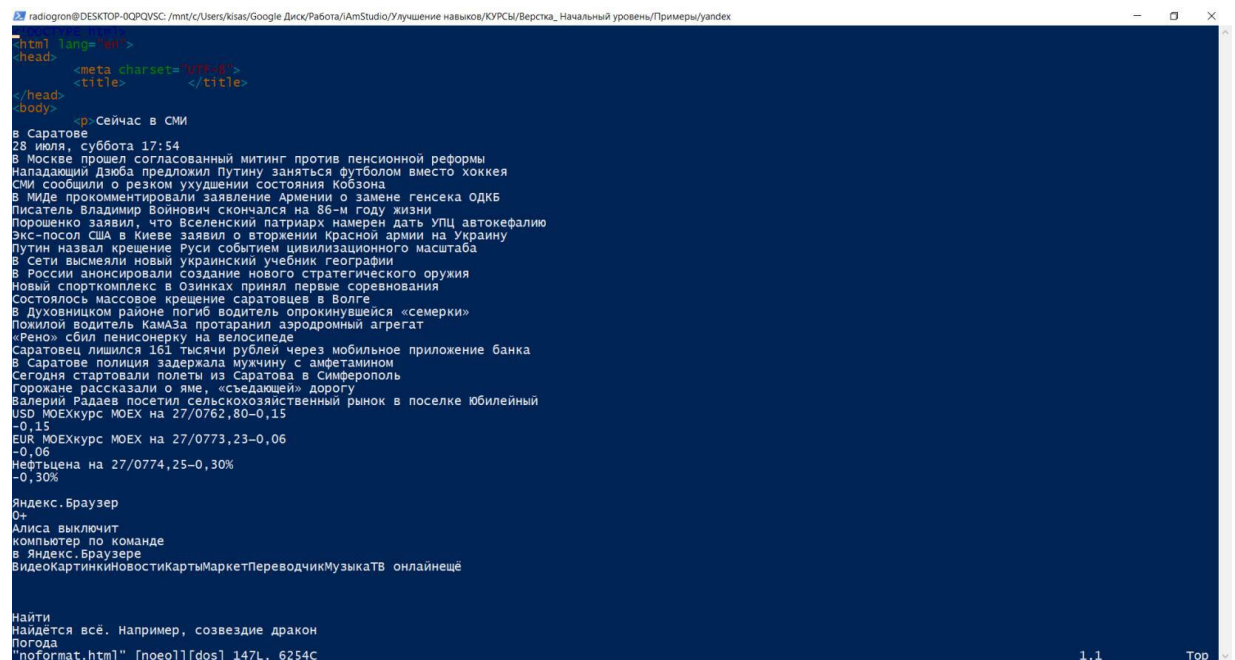
Редакторы кода

Выбор редактора кода (IDE)

Прежде чем мы приступим к практике, вам необходимо выбрать редактор. Иными словами IDE. IDE – это интегрированная среда разработки. Иными словами – среда в которой разработчик (тут верстальщик может почувствовать гордость, что он на одной ступени с программистами) пишет код и ему в этом помогают всякие утилиты, виджеты и прочая вспомогательная ерунда. Если у вас подсвечивается код в соответствии с синтаксисом выбранного языка – это IDE.

Но будем откровенны. Изначально мы работали в редакторах исходного кода. Мы это верстальщики, программисты, все, кто работал с кодом. Таким редактором долгое время был, и наверное остался блокнот, который поставлялся и поставляется до сих пор с операционной системой Windows. Со средой разработки в Linux не все так просто. Там есть Vim – редактор, позволяющий обрабатывать содержимое файлов прямо в режиме консоли.

Если открыть в Vim файл формата html с примером кода, описываемого в главе Теги, то можно увидеть вот это:



```
radiogron@DESKTOP-00PQVSC: /mnt/c/Users/kisas/Google Диск/Работа/AmStudio/Улучшение навыков/КУРСЫ/Верстка_ Начальный уровень/Примеры/yandex
<html lang="ru">
<head>
  <meta charset="utf-8">
  <title>
    </title>
</head>
<body>
  <p> Сейчас в СМИ
в Саратове
28 июля, суббота 17:54
8 Москве прошел согласованный митинг против пенсионной реформы
Нападающий Дзюба предложил Путину заняться футболом вместо хоккея
СМИ сообщили о резком ухудшении состояния Ковбона
В МИДе прокомментировали заявление Армении о замене генсека ОДКБ
Писатель Владимир Войнович скончался на 86-м году жизни
Порошенко заявил, что Вселенский патриарх намерен дать УПЦ автокефалию
Экс-посол США в Киеве заявил о вторжении Красной армии на Украину
Путин назвал крещение Руси событием цивилизационного масштаба
В сети высмеяли новый украинский учебник географии
В России анонсировали создание нового стратегического оружия
Новый спорткомплекс в Озінках принял первые соревнования
Состоялось массовое крещение саратовцев в Волге
В духовническом районе погубил водитель опрокинувшейся «семерки»
Пожилой водитель камаза протаранил аэродромный агрегат
«Рено» сбил пенисонерку на велосипеде
Саратовец лишился 161 тысячи рублей через мобильное приложение банка
В Саратове полиция задержала мужчину с амфетамином
Сегодня стартовали полеты из Саратова в Симферополь
Горожане рассказали о яме, «съедающей» дорогу
Валерий Радаев посетил сельскохозяйственный рынок в поселке Юбилейный
USD MOEXкурс MOEX на 27/0762,80-0,15
-0,15
EUR MOEXкурс MOEX на 27/0773,23-0,06
-0,06
Нефтьцена на 27/0774,25-0,30%
-0,30%

Яндекс. Браузер
0+
Алиса выключит
компьютер по команде
в Яндекс. Браузере
видеоКартинкиНовостиКартыМаркетПереводчикМузыкаТВ онлайнещё

Найти
Найдётся всё. Например, созвездие дракон
Погода
"noformat.html" [noel] [dos] 147L, 6254C
1.1 Top
```

Редактор Vim в консоли Windows PowerShell с установленной подсистемой Ubuntu

Никто не знает, как выйти из вима

Сей факт породил тонны шуток и мемов в духе:



На самом деле все просто, достаточно ввести двоеточие и символ q в английской раскладке. Никакой интриги.

Язык не поворачивается назвать Vim IDE, так что я скорее отнесу его к редактору исходного кода. К слову, если блокнотом сейчас никто не пользуется, то Vim более чем здравствует. Если вы начнете использовать Linux систему, или, как я Windows, с установленной подсистемой Ubuntu и будете использовать систему хранения версий, то скорее всего столкнетесь с этим редактором.

Минутка истории

Самым первым редактором исходного кода можно считать что угодно, вплоть до гальки на песке, которая описывает алгоритм. Но более близкое к нашему времени можно считать дырокол (вспоминаем перфокарты) и лист бумаги\тетради. Именно с использованием обычной аналоговой бумаги мои родители писали и сдавали

программы во время обучения в институте.

Но вернемся к вопросу выбора. Выбор редактора (возьмем в качестве этого термина объединение IDE и редактор исходного кода) очень важен как для верстальщика, так и для программиста.

Я уверен, в первую очередь он должен быть удобным. Для меня удобство – это качественные шрифты и цветовая схема, которая не вырывает мне глаза. Лучшим редактором в плане цветового решения для меня был и остается Notepad++. Там информация отображается на белом фоне, по-умолчанию используется хорошо читаемый шрифт. Но он сильно пасует перед продвинутыми редакторам в плане возможностей, хотя я достаточно часто к нему прибегаю, когда необходимо решить сложные вопросы легким путем.

Полноценная IDE, на которой я остановился на момент написания этой книги – Brackets. Это достаточно удобный редактор для работы над проектами. Когда много файлов, и нужно все их видеть не отходя от кассы, т.е. от редактора. В тот же notepad такую возможность ввели где-то в середине 2018-го года.

Иной раз и при особых задачах пасует и Brackets. Тогда на помощь может прийти WEBStorm. WEBStorm создала компания JetBrains и у них есть еще много редакторов под разные нужды. В основном эти нужды выражены языками.

В целом можно выделить следующие редакторы:

Notepad – Встроенный редактор Windows. Никакой подсветки, открывается очень быстро, позволяет обнулять стили.

Notepad++ (NPP) – Грубо говоря, расширенный редактор Notepad, позволяет полноценно редактировать код, и мы все когда-то с ним успели поработать.

Sublime – Редактор из Linux, настолько знаменит, что о нем не все знают. В целом, удобная альтернатива Notepad++. Или наоборот!? Рекомендую ознакомиться.

Dreamweaver – Хоррор из прошлого. Некогда запомнился мне инновационной подачей «пишу и вижу», что позволяло сразу видеть, как код написан. Вот только показывал он его как-то по своему, в своей реальности. Лет 7 назад посоветовал бы не связываться с ним, а что он представляет сейчас не знаю.

Brackets – Мощный и удобный редактор для верстальщика. Является неким средним явлением между notepad++ и phpstorm. Можно ставить дополнения и вообще достаточно гибок в настройке.

Visual Studio Code – Я когда-то учил язык C и тестировал знания в редакторе Visual Studio. Так вот с тем, к чему я привык, там эта «облегченная версия» не имеет ничего общего. Удобный расширяемый редактор, который я поставлю на одно место с brackets. По умолчанию умеет работать с Git.

PHPStorm – Ориентирован изначально на работу php-программиста, но отлично подходит и верстальщику. Удобная работа с проектами, некоторые интересные фишки типа редактирования открывающего и закрывающего тега. Но не умеет редактировать много строк сразу, да и тяжелый, как мастодонт. Будет грузиться едва ли меньше, чем фотошоп на вашем компьютере. Рекомендую только в том случае, если вы работаете в паре с php-программистом и вам не обойтись без единовременного редактирования кода.

WebStorm – Редактор с упором на javascript проекты. Должен быть удобен для возведения конструкций с использованием фреймворков/библиотек типа Vue, Angular, React.

PyCharm – Это уже третий редактор от JetBrains, и он вроде не по теме, но только до того момента, как вы не начинаете работать с, допустим, фреймворком Django. Это уже относится к продвинутому уровню, так что не будем долго останавливаться.

Остальными критериями для выбора редактора я считаю расширяемость и в принципе возможности, о которых не думаешь в первую очередь. Например в sublime и brackets можно редактировать несколько строк одновременно.

Ни в коем случае не используйте для верстки визивиги!

WYSIWYG – это визуальный редактор, работающий по принципу «что видишь, то и получишь». Такой редактор уместен во многих других направлениях, например для сбора презентации в Point, или при составлении текста в Google Docs, но только не в верстке. Уже придумали, но еще не создали такой визивиг, который мог бы вам помочь в жизни.

К широко распространенному визивигу можно отнести редактор в WordPress. Представьте ситуацию: вы написали шикарный код, читаемый, с отступами, как надо, вам необходимо разместить его на

страницу, идете в редактор вордпресса, вставляете, сохраняете. Потом какой-то очень умный человек переключается в текстовый режим и обратно. Все, код поехал.

Настройка IDE

Эти действия я буду описывать на личном примере с использованием редактора Brackets.

Одно из первых действий, что я делаю при установке чистой программы – качаю тему. Часто скин под Notepad++, иногда темную тему, иногда вообще что-то новое.

Далее ставлю Emmet – данное дополнение позволяет использовать краткую форму записи тегов.

Сокращенный код в Emmet: `div.classname#ddd>label+input`

Преобразованный в HTML: `<div class="classname" id="ddd"><label for=""></label><input type="text"></div>`

И в обязательном порядке устанавливаю плагин Show Whitespace в исполнении Dennis Kehrig (В магазине несколько таких плагинов и этот, на мой взгляд, самый адекватный в исполнении). Этот плагин позволяет подсвечивать пробелы и табуляцию – таким образом круче управлять своим кодом.

После установки плагинов я устанавливаю настройку отступов в формат табуляции со значением в 4.



Настройка находится в нижнем правом углу редактора Brackets.

Лично мне легче работать с кодом именно таким образом. Так отступы всегда формируют читаемый код, что особенно важно при работе, например с препроцессором SASS (не путать с SCSS). Другой популярный метод – пробелы в 2 единицы.

Настраивать и выбирать вам, но так как вам либо работать в команде, либо отдавать свой код заказчику, с которым работать потом другому верстальщику или программисту, придерживайтесь культуре

написания кода и делайте код читаемым хотя бы с помощью адекватных отступов.

Это в общем все. Для комфортного написания кода стоит просто настроить свое пространство для этого самого комфорта. К теме также можно отнести и выбор хорошего кресла, но это уже совсем другая история.

Не забывайте, кроме фактора удобства и популярности существуют требования языка. Например YAML требует запись в 2 пробела.

Когда спасает Notepad++

Автозамена в 20+ файлах – это просто волшебный редактор, когда необходимо провести автозамену в тонне файлов, любой другой навороченный редактор заглохнет и начнет чихать.

Редактирование отдельно-стоящих файлов. Незачем грузить полноценную IDE для редактирования одного файла, например при использовании FTP-клиента. На всех компьютерах, на которых я работал, notepad загружается крайне быстро. В то время, как тот же Brackets приходится ждать. Чего уж говорить про PHPStorm.

Необходимо обнулить стили. Если я беру какой-то текст с сайта, то вместе с ним тянутся и стили, заголовки и иное форматирование. И не всегда связка Ctrl+Shift+V спасает. В таком случае я запускаю Notepad ++.

Палочка-выручалочка

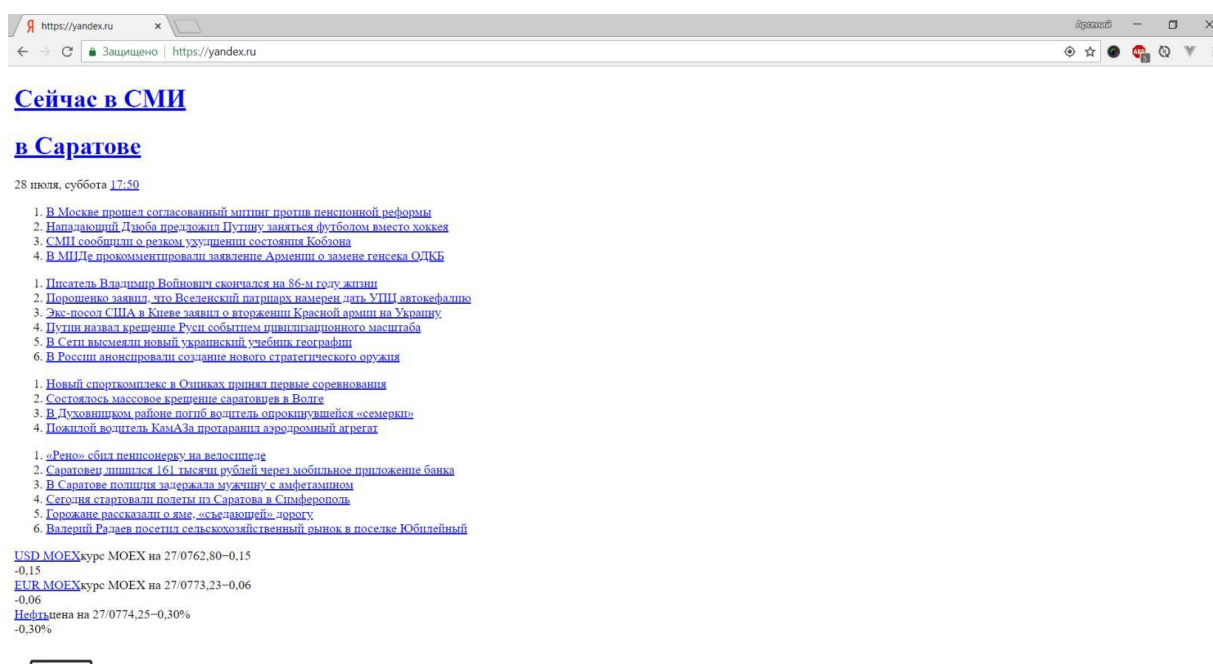
Если вы открыли ту самую тонну файлов в notepad и пытаетесь их закрыть, в меню Файл есть волшебная кнопка «Заккрыть все».

HTML

Теги

Тег HTML – это уже и есть HTML. Теги используются для того, чтобы обозначить начало и конец элементов на будущей странице. Использование разметки изначально необходимо для того, чтобы облегчить читаемость.

Для примера я решил взять главную страницу Яндекса:

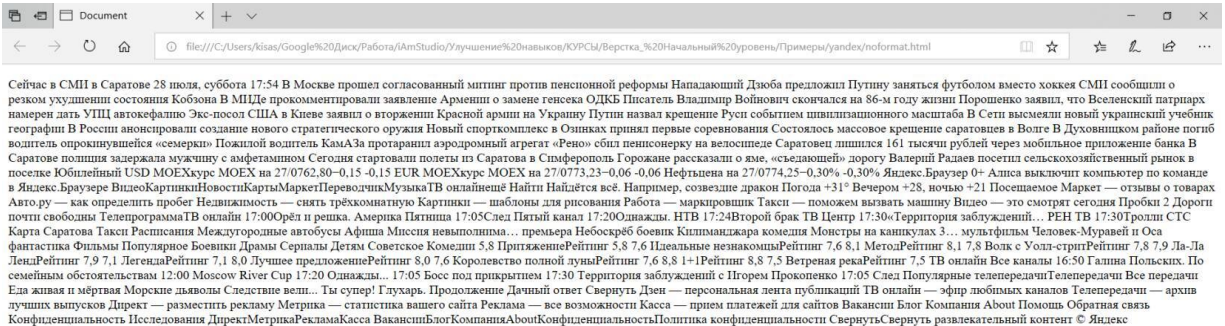


Главная Яндекса без стилей.

Для наглядности я удалил тег `head` вместе со всем содержимым. Внутри были стили и скрипты. И много всякой полезной ерунды, без которой мы можем видеть отформатированный, де-факто сверстаный текст. В данном случае текст и информацию вывел сервер, так что если вы сделаете то же самое у себя на компьютере, то увидите другой текст на главной странице.

Важно!: кроме `head` я удалил шапку и содержимое, которое выводило мою личную информацию.

Вот то же самое, только без участия тегов. Все содержимое я скопировал и вставил в тег p.



Главная Яндекса без верстки вообще.

Внимательный читатель обратит внимание, что 2 страницы браузера не сильно похожи одна на другую. Таким образом можно утверждать, что верстка – это перевод текста [читай контента] в другое качественное состояние. Точно так же, как это делают в типографии в классическом исполнении верстки.

В данном примере используются теги заголовков (h1), параграфа (p) и нумерованного списка (ol).

Обратите внимание:

Все теги форматирующие текст по-умолчанию имеют свое оформление. Например у списка есть отступ слева + нумерация, заголовки отличительно больше, чем основной текст, ссылки подсвечиваются и подчеркиваются и т.д. Первое желание начинающего дизайнера – поменять все стили на какие-нибудь «крутые». Ссылки сделать в цвет радуги, заголовкам дать новые отступы или учудить еще какую-нибудь ересь. Окститесь, ребята, в большинстве случаев не

стоит так делать. Для того и создавались стандарты, чтобы пользователь мог понять, что ссылка это ссылка, заголовок это заголовок, который относится именно к вот этому абзацу. В этом деле существует много исключений, но работая с текстом всегда нужно взвесить все за и против, прежде, чем необдуманно броситься делать все «красивенько».

К слову, теги используют не только для верстки сайтов, но и для оформления текста, например в вики-разметке раньше существовали (сейчас скорее всего тоже где-то используются) bb-коды, созданные для оформления текста на форумах.

Сегодня, с ростом качественных редакторов типа zen-editor, мы все реже видим и используем теги в их первозданном виде при обычном редактировании текста в веб-интерфейсах. Но не обманывайтесь, под капотом редактора происходит оборачивание текста в теги.

HTML дескрипторы

Специально для написания этой книги, кроме знаний, которые я получил опытным путем в разработке разного рода проектов, я активно ищу теоретическую информацию. Таким образом я узнал, что слово тег – это сленг, который используется вместо термина дескриптор.

Еще одним важным примером будет XML разметка. Она вам пригодится при разработке сайта на продвинутом уровне, например при создании священного файла sitemap.xml. Священный он потому что при его отсутствии на сайте поисковики будут ставить вам палки в колеса.

Идя дальше можно говорить про разработку для мобильных платформ. Там вам тоже пригодится знание и понимание тегов.

Как применять теги

Существует 2 вида тегов. Те, которые надо закрывать и те, которые закрыты сами. Отличить их просто, те, что надо закрывать одинаковые, идут парно и последний, т.е. второй имеет одну отличительную черту. Косую. Закрываемые теги выглядят так:

<h1>Между тегами содержимое</h1>

Открывающий тег / Закрывающий тег с косой чертой

В то же время есть самозакрывающиеся теги. Их также называют одиночными, пустыми. И если обычный тег надо открыть и закрыть, то одиночный необходимо просто разместить.

**
**

Вот и все

Открывающий тег, он же закрывающий

На фоне парных тегов количество одиночных весьма ограничено, и их легко запомнить. В разборе значений и предназначений тегов я разделяю парные и одиночные теги на 2 группы.

С приходом стандарта HTML5 изменилась форма записи в одиночных тегах. Раньше требовалось указывать косую черту перед закрывающей скобкой тега вот так – `
`, а сейчас нет. Это долгое время мешало мне, так как я привык писать по старым стандартам, и валидатор W3C писал мне предупреждения.

Теги можно спокойно, с оглядкой всего-лишь на правила, вкладывать друг-в-друга. Самым простым примером должна стать ссылка внутри абзаца:

**<p>Текст, внутри которого находится ссылка
</p>**

Пример вложенных тегов

Кроме вида тегов к ним применимы атрибуты. Атрибут тега – это свойство, которое может иметь значение присвоенное, или просто свойство, которое не имеет значения, но определяет, как будет себя вести на странице тот или иной тег.

Основные теги

Тегов существует великое множество и нет никакого смысла их все запоминать. Для постоянной работы вам потребуется несколько основных: `div`, `span`, `p`, `a`, `img`.

Также обращаю ваше внимание, что данный список не является конечным, так как для повышения семантики и для удобства верстки их дополняют, а многие выходят из обихода. Подробный список можно найти на сайте w3schools.com/tags/.

Парные теги

Самые первые парные теги, которые вам нужно запомнить – те, которые формируют каркас будущей страницы:

html – Данный тег определяет страницу целиком. В него оборачивается все содержимое страницы.

head – Данный тег предназначен для «невидимой» части страницы. В него помещается заголовок страницы (который отображается на вкладке браузера), определение правил и почти всегда ссылки на зависимости.

body – Само тело [body] содержит видимую часть страницы, грубо говоря отформатированный текст.

Тут же следует сразу показать, как их использовать:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Пример</title>
  </head>
  <body>
    Содержимое
  </body>
</html>
```

Пример использования парных тегов

Первым делом мы указываем доктайп, затем открываем html тег, в котором размещаем head внутри которого можно разместить title. Закрываем и тут же открываем body, в котором уже пишем весь код. В конце теги закрываются в порядке вложенности.

Маленький хак от Emmet

Возвращаясь к теме редакторов IDE можно упростить себе жизнь с плагином Emmet. Он позволяет вызвать шаблон-заготовку страницы html посредством следующих действий:

Пишем знак восклицания – !

Нажимаем таб

Получаем кусок кода, готовый к эксплуатации

Я такое часто применяю в редакторе Brackets.

Далее давайте разберем остальные теги.

<div> – Самый популярный тег, который используется для формирования страницы. Является блочным по-умолчанию. Т.е. растягивается по всей ширине занимаемого пространства и сразу понимает присвоенным значения. Например, если тегу а указать размеры, то он их не сможет понять до тех пор, пока вы в стилях не укажете ему другой display, например inline-block. Div это умеет делать сразу.

<h1>–<h6> – Тег заголовка [англ. heading – отсюда сокращение h], варьируется от уровня заголовка. H1 – самый важный и, соответственно, самый большой. А h6 уже меньше размера основного текста.

Заголовки жизненно необходимы для разделения текста на части, чтобы упростить чтение материала. Они также используются поисковыми роботами для определения содержимого страницы. Впрочем, и человек в первую очередь пробегаются глазами по заголовкам и только потом читает сам текст (если читает).

Заголовок является блочным элементом.

На странице может быть не больше одного тега h1!

<p> – Данный тег используется для оформления текста в формате абзаца [англ. paragraph]. Чтобы абзац выглядел, как должен, а именно: быть единым целым, единицей текста, и иметь перенос текста в конце. Текст без обертки в параграф превращается в однородное полотно, которое трудно читать. Параграф не может быть обернут в тег ссылки.

**** – Этот тег также представляет единицу текста, оптимально подходит для конструктивного (визуально никак не выделяется) выделения текста для каких-либо манипуляций внутри абзаца, или для существования отдельно, может быть размещен внутри ссылки. Не может оборачивать блочные элементы, в том числе заголовки.

<a> – Ссылка это также единица текста, которая отвечает концепции гипертекстовой разметки, позволяет ссылаться на якоря страницы (без перехода), либо на другую страницу сайта или в принципе на другой сайт.

Для данного тега обязательным атрибутом является указанная ссылка href. Если его не указать, то это будет такой же обычный текст, как и span.

Т.е. корректная ссылка должна выглядеть так:

`Ссылка на яндекс` – К ссылкам также применима семантика, которая будет описана позже.

``, `` – Тег используется для выделения текста жирным/bold. Можно с легкой душой использовать внутри тега `<p>`, так и отдельно стоящими.

Обращаю ваше внимание на то, что `b` является декоративным тегом, в то время, как `strong` не только выделяет текст жирным, но и делает его более важным с точки зрения обработки контента поисковым пауком.

`<i>`, `` – Также, как и `` делает текст выразительным, а именно выделяет слова или фразы курсивом. Может использоваться внутри тега `<p>`, но и отдельно. Точно также, как с предыдущим тегом `i` делает текст просто наклонным, а `em` в свою очередь ориентирован и на поисковики.

Стараниями дизайнеров и верстальщиков тег `<i>` стал повсеместно использоваться для размещения иконок из набора иконочных шрифтов. Самым популярным на момент написания книги является FontAwesome 5. И вставляется на страницу он так:

`<i class="fas fa-play"></i>`

`<strike>`, `` – История точно такая же, как и с предыдущими двумя тегами, только этот зачеркивает слова. Вот так – зачеркнутый текст. И как и с предыдущими примерами `strike` несет важность для поисковиков.

Все три тега можно комбинировать. Например, чтобы получить зачеркнутый жирный текст курсивом можно сделать так:

`<strike><i>Текст</i></strike>`

`<sup>`, `<sub>` – Данные теги сейчас используются редко и преимущественно в качестве специфичных. Они нужны для того, чтобы отобразить содержимое в верхнем или нижнем индексе. А выглядит это так:

Sup – Текст верхний индекс

Sub – Текст нижний индекс

Оптимально использовать для обозначений формул. Классическим примером является форма записи химического состава воды H₂O, в то время, как верхний индекс лучше применять для сносок.

`<small>`, `<big>` – *small* делает текст меньше, а *big* наоборот. Важно не забыть, что меньше или больше понятие относительно и чтобы

корректно работали эти два тега – они должны быть внутри текста с размерами. Проще всего представить его внутри тега *p*.

<mark> – Данный тег необходим для выделения текста, для подсветки. Результат его можно сопоставить с подкрашиванием текста желтым фломастером.

<header> – Этот тег и несколько следующих были введены вместе со стандартом HTML5 и они нужны непосредственно для корректной семантической верстки. Ранее мы спокойно обходились дивом с классом *header*, чтобы визуальное его оформить. Сейчас есть *header*, который можно использовать для оформления шапки страницы.

<footer> – Противоположный тег *header*. Нужен для логического выделения подвала как всей страницы, так и для подвала секции.

<main> – Использование этого тега предполагает размещение в нем основного содержимого. Всей страницы сайта. Я по умолчанию применяю его в разметке между *header* и *footer* при верстке макета сайта.

<aside> – Этот тег по смысловой нагрузке противоположен тегу *main*. В нем должна находиться вспомогательная информация, которая ранее часто располагалась слева или справа от основного содержимого сайта. Иными словами – сайдбар.

<article> – Данный тег определяет, что внутри него находится информация, которую можно расценивать, как статью. Отличительными составляющими статьи являются: заголовки, текст.

Если вы хотите как-то выделить, например анонс новости используйте *section*. В то же время W3C рекомендует использовать данный тег для оборачивания комментариев. Их следует воспринимать как отдельное произведение мысли.

Если вы используете комментарии в тексте, вы можете обернуть их в тег *article* внутри *article*.

<section> – Я так и хочу назвать этот тег *article* в миниатюре. Он нужен для выделения на странице логических блоков. Например им можно выделить главу книги, как я указал выше – анонс статьи, новости, набор заголовков. В общем и целом легко и спокойно можно вместо использования тега *div* ставить этот.

<nav> – Тег выделяющий навигацию на странице. Крайне важный для роботов, так как помогает им при навигации по сайту.

Разумно использовать его для оборачивания основной навигации в шапке сайта, в подвале. Я видел мало случаев, но его очень уместно использовать для любой навигации, которая состоит из более, чем одной ссылки. Хороший пример – список вспомогательных страниц в сайдбаре, или переключение между «предыдущей» и «следующей» страницей в блоге в конце статьи.

<iframe> – Хотите открыть сайт в сайте? Пожалуйста.

Вы можете спокойно открывать ссылки и файлы верстки используя этот тег. Достаточно указать размеры и ссылку, которую нужно открыть.

Давно таким образом имитировали асинхронность сайта, была навигация в шапке сайта, а под ней фрейм. Нажимая на ссылки вы просто открывали новую ссылку в фрейме. В то же самое время окно браузера, которое вы открыли не перезапускалось. Возможно из-за этого в свое время и стали порицать использование фреймов, да так, что сейчас отыскать по ним информации можно не так уж и много.

Но спешу вас убедить, они более чем живы и используются полным ходом. Просто об этом мало говорят. Видели чат на каком-нибудь сайте, например *jivosite* или чат с группой вконтакте? Он сделан с помощью фрейма. И таких примеров много.

Но это не должно стать примером того, что фреймы надо использовать часто и много. Совершенно нет, к ним надо подходить с умом и только там, где действительно это необходимо.

****, **** – Данные два тега используются для оборачивания списка.

ol – нумерованного (ordered list), а **ul** – маркированного (unordered list). Внутри них элементы списка уже оборачиваются в *li*. По умолчанию эти теги имеют оформление в виде отступов слева, для маркера или номера. Маркер может находиться как внутри элемента, так и снаружи. Часто используется для оформления навигации и в таком случае конструкцию необходимо обернуть в тег *nav*.

**** – Элемент списка. Имеет визуальное оформление похожее на параграф – с отступом вниз и по умолчанию подчиняется стилям текста присвоенным самому верхнему уровню (*body*, *html*). Непосредственно в него и размещается содержимое. Оно может быть любым из строчных (инлайновых), будь то текст или картинка.

Непосредственно в этот тег можно вложить еще один уровень списка (вложенный список).

<pre> – Предназначен для вывода содержимого предварительного форматирования. Это значит, грубо говоря, как написал, так и выводится. В другом теге если вписать несколько пробелов подряд, при открытии верстки в браузере большинство пробелов исчезнет. При использовании этого тега нет.

Самый популярный случай использования этого тега – для вывода примеров кода.

А раньше его часто использовали, чтобы вывести текстовую картинку. Да, была такая забава:

```
_____Oooo_____  
____oooO____(____)  
____(____)____/_____  
____\____(____/_____  
____\____)_____  
_____
```

<code> – Формально этот тег нужен непосредственно для вывода кода, вот только он не учитывает отступы и пробелы так, как это делает pre. Но ничего не мешает использовать их совместно для достижения необходимого результата.

Хотя сегодня при использовании примеров кода и уместнее задействовать, к примеру jsFiddle.

<marquee> – Не хватает бегущих строк в рекламе на улицах – вот вам возможность реализовать то же самое в браузере. Я бы очень не рекомендовал использовать этот тег, хотя с его помощью можно креативно решать проблемы поставленные перед верстальщиком.

Последний раз я его использовал году эдак в 2012-м, когда делал анимацию облаков на сайте. Я заставлял их двигаться справа налево просто вставив картинки в бегущую строку.

Вообще в этот тег можно засунуть практически в что угодно и заставить его прокручиваться не только по горизонтали, но и по вертикали.

<blockquote> – Тег для выделения цитат. Подходит для использования цитат отдельным блоком на странице, для длинных цитат. По умолчанию выделяется на фоне основного текста отступами.

<cite> – Тоже цитаты, но выделяются наклонным текстом и уместнее использовать их для коротких цитат внутри текста или

сносок.

Хотя так и хочется использовать только cite для выделения цитат, название вроде бы говорящее, но я рекомендую с мозгами подходить к данному вопросу.

<nobr> – Этот тег можно рассматривать, как противоположный принудительному переносу строки br. Но, в отличие от последнего, вызовет проблемы при валидации кода.

Предназначен для принудительного НЕ переноса текста. Уместно использовать для названий, которые обязательно должны идти вместе, например «ООО Флюгегехаймен». Если после ООО при загрузке страницы будет перенос, потому что так открылось, то и выглядеть это будет некорректно, и, по сути, являться ошибкой правильного написания названий.

Поэтому в таких случаях я продолжаю использовать этот тег, несмотря на выход из стандартов.

Будьте аккуратны, так как он может дать горизонтальную прокрутку. А ее никто не любит. Только экстравагантные дизайнеры.

<table> – Табличная верстка достаточно маленькая, поэтому можно описать их кратко. Чтобы начать описывать таблицу используйте этот тег. Все просто.

К нему можно применять атрибуты, такие как толщина линий, border="1px".

<thead> – Логический блок означающий начало таблицы, ее шапку. В него уместно помещать строки содержащие заглавную информацию. Хороший пример – в этой таблице в самом верху, в первой строке есть 2 ячейки: «Тег» и «Описание». Чтобы сверстать таблицу можно обойтись без него.

<tfoot> – То же самое, что и thead, только для конца таблицы. Хорошо подходит для выводов, сумм значений таблицы во всяких отчетах.

<tbody> – А вот этот необходим для основного содержимого таблицы.

<tr> – Таблица состоит из строк и ячеек. Не бывает колонок, только строки.

<td> – Самая маленькая составляющая таблицы – ячейки. Обладают свойствами пожирать рядом находящиеся ячейки по горизонтали и по вертикали (строками).

Эти свойства:

Colspan – на сколько ячеек вширь распространяется описываемая ячейка.

Rowspan – на сколько строк распространяется ячейка.

<form> – Чтобы создать форму чего-либо стоит использовать этот тег. В него помещаются все поля ввода, такие как `input` и `textarea`.

А на саму обертку обычно вешаются обработчики типа `action=POST`. Обработка форм сложная тема и плавно перетекает в программирование, например PHP. В область которого попадают данные, введенные пользователем.

<fieldset> – Блочный элемент необходим для группировки полей ввода, например несколько инпутов типа чекбокс. Имеет оформления в виде границ.

<legend> – Данный тег позволяет ставить заголовки внутри тега `fieldset`. Он помогает пользователю легче ориентироваться на странице.

<textarea> – В отличии от `input`, который рассчитан на ввод текста в одну строку, этот тег используется для ввода многострочного текста. Если не задать ему размеры, то при наборе текста он будет становиться больше, подстраиваясь под количество введенных строк.

<button> – Тег для кнопки, который используется в форме. В отличии от `input type="button"` позволяет разместить в себе, например картинку. А в `input` только текст.

Одиночные теги

Одиночные теги отличаются от парных, как я указал ранее тем, что у них нет закрывающего тега.

**** – Показать изображение просто: достаточно разместить реальное физическое внутри файлов сайта или взять ссылку на любую картинку в интернете и указать ее в атрибуте src. На выходе вы получите встроенную в страницу картинку в реальный размер.

Дальше начинается самое интересное – по настройке изображения таким образом, чтобы оно правильно вписывалось в страницу. Но это уже совсем другая история.

**
** – Тег принудительного переноса строки. Если вы хотите, чтобы фраз

ы обрыва

лись так, используйте этот тег в конце каждой строки.

<hr> – Горизонтальная черта. По умолчанию имеет отбивку сверху и снизу. А по смысловой нагрузке позволяет отделить часть информации друг-от-друга. В принципе, в том же контексте, порой уместнее использовать негативное пространство.

<input> – Поле ввода широкого спектра действия. Его применение зависит от задачи, которая стоит перед верстальщиком.

Чтобы это поле ввода работало необходимо указать его тип. А их у него существует много:

1. Text – Для ввода обычного текста
2. Email – Для ввода почтовых адресов
3. Password – По умолчанию заменяет введенный текст на звездочки
4. Button – Аналог тега button, но в форме лучше использовать последнее

5. Submit – Выглядит также, как button, но если использовать внутри тега form, то нажав на него пользователь запустит обработку формы

6. Reset – тоже применяется в формах, для очистки полей.

7. Checkbox – Этот тип используется для предоставления пользователю возможности указать один или несколько пунктов в форме. Например, если нужно выбрать несколько свойств товара: допустим, к покупаемым очкам вы хотите добавить опционально коробочку, салфетку для протирания.

8. Radio – В отличие от чекбокса позволяет выбрать только один вариант из группы, на примере тех же очков это будет цвет оправы. Сами по себе эти два типа не могут идти отдельно от текста, так как выглядят сами по себе не интуитивно.

9. Image – Используется для загрузки изображения с компьютера. Обращаю внимание на то, что само по себе это поле только сохраняет ссылку на картинку, а не саму картинку. Для этого необходима обработка этой ссылки скриптом.

10. Hidden – Скрытое поле. Используется для скрытой обработки и передачи данных скрипту, обрабатывающему форму, например тем обращения, или маркер валюты.

Данный список не полный, но достаточный для того, чтобы понять, как именно верстать.

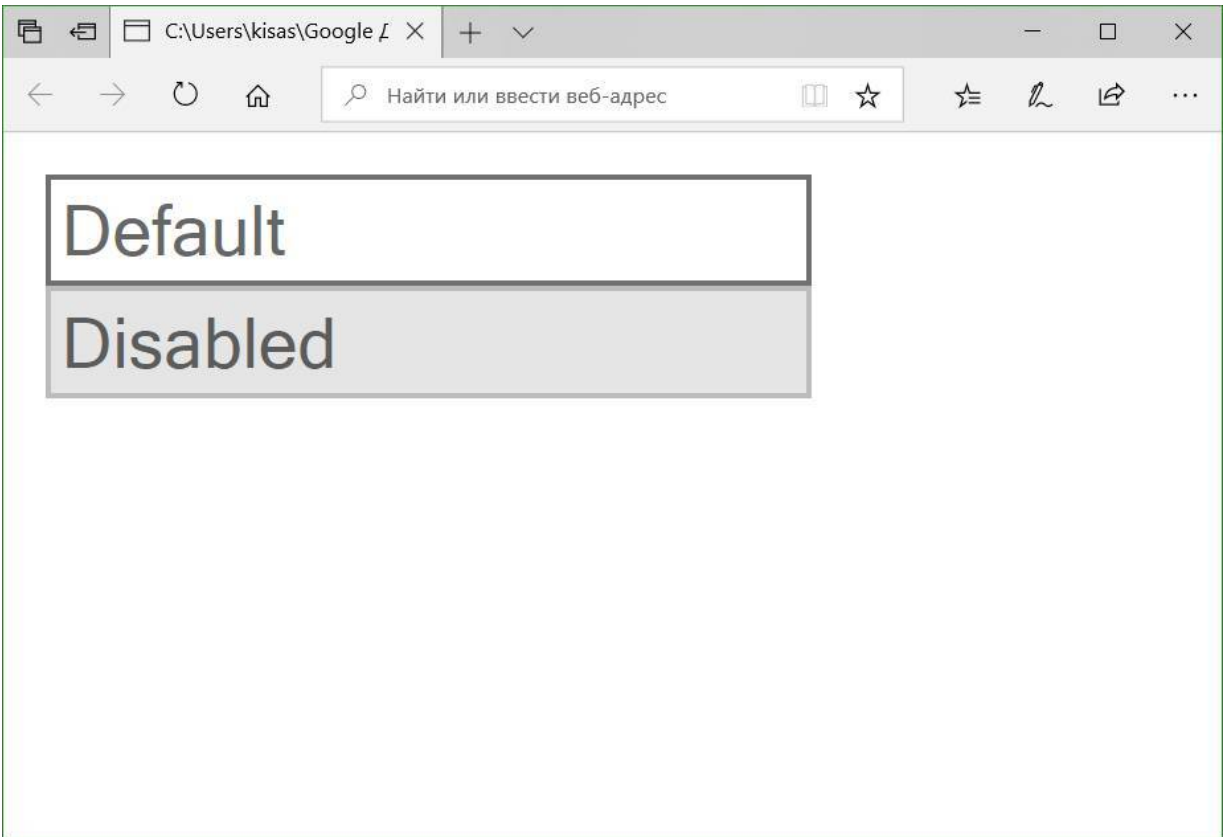
Атрибуты

Чтобы сделать жизнь верстальщика еще сложнее были придуманы атрибуты, которые существенно влияют на то, как работают теги. Но не стоит бояться, после того, как вы разберетесь в теме – не составит особого труда понять, что и как с ними делать.

Существуют универсальные теги, и те, которые привязаны к определенному тегу. К универсальным можно отнести *hidden*, который прячет элемент на странице, а к частному с привязкой – *action*, который работает с тегом *form*.

```
<form action="get.php" method="get" class="action-form">  
  <input type="text" placeholder="Default">  
  <input type="text" placeholder="Disabled" disabled>  
  <input type="text" placeholder="Hidden" hidden>  
</form>
```

Пример использования универсальных парных и одиночных атрибутов



Работа тегов в браузере

Также следует обратить внимание на атрибуты, которые должны нести значения и те, которые влияют на тег просто своим присутствием. К первому можно отнести как `action` из примера выше, так и `class`, который содержит в себе идентификатор тега.

При этом тот же *hidden* или *disabled* влияет на тег. Первый скрывает его страницы, другой отключает возможность взаимодействовать, например, с кнопкой или полем ввода.

Еще раз о том, как использовать теги

Теперь, когда у вас есть представление о тегах, которые я классифицирую как основные и наиболее часто используемые, необходимо отметить тот факт, что секция как применять теги была только предварительными ласками.

Я рассказал, как использовать парные и одиночные теги, но в современной верстке уже почти нельзя встретить чистое их использование. Исключение составляют разве что одинарные, такие как `b` или `h`. В остальных случаях активно используются атрибуты.

Самыми популярными и важными атрибутами являются `id` и `class`. Раньше, до того как ввели классы, мы использовали `aid`. Основная проблема заключалась в том, что `aid` – это уникальный идентификатор. Он может быть только один на странице, в то время, как классом обозначают однотипные элементы и их может быть много. А так как он может быть только один, то нельзя было писать стили для группы элементов. Классы сильно выручают верстальщика.

Чтобы использовать тег с атрибутом необходимо использовать следующую схему:

```
<p class="класс">Содержимое</p>
```

Пример использования атрибутов типа класс.

На данном примере видно, что происходит открытие тега скобкой `<`, затем указывается сам тег, в данном случае это параграф `p`, затем до закрытия скобки `>` указываются все атрибуты. После уже размещается содержимое – текст, после чего тег закрывается парным закрывающим.

В этом примере я показал, как присваивать класс, точно таким же образом можно вместо атрибута `class` указать `id`, или любой другой.

Также могут быть атрибуты без значения, например если нужно указать, что поле типа `input` обязательно к заполнению. Тогда указывается таким образом:

```
<input type="text" required>
```

Пример поля ввода обязательного для заполнения.

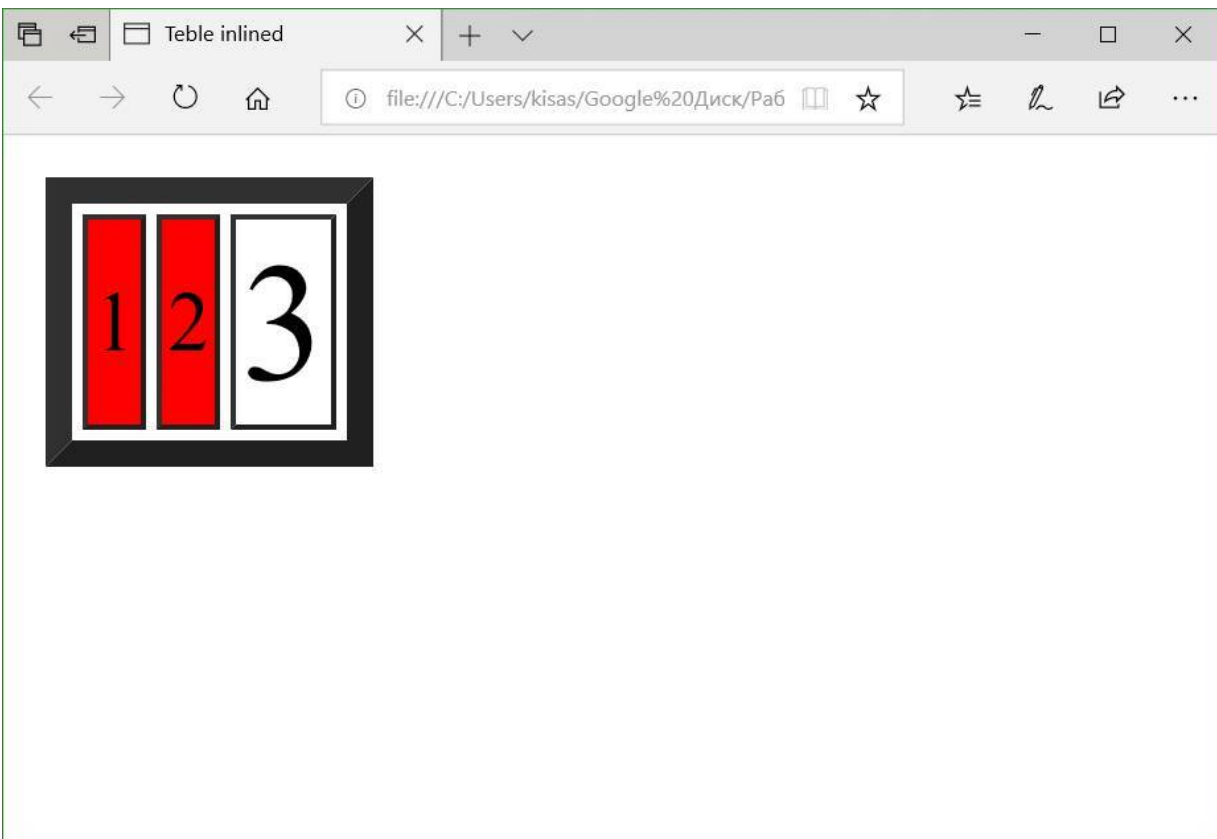
Если вы попытаетесь отправить форму с незаполненным полем, у которого стоит атрибут `required`, то браузер остановит вас и потребует вернуться к заполнению. Такие вещи можно отнести к базовой валидации форм.

Инлайновые стили

Раньше мы часто использовали инлайновые стили. До того, как в обиход вошли каскадные стили, это был единственный способ создания визуально отличительных черт блоков – писать инлайновые стили.

Инлайновые стили – это создание визуального оформления средствами атрибутов и написания стилей в атрибуте style.

Чтобы описать визуализацию таблицы, например таким образом:



Можно использовать следующий код:

```
<table border="5">
  <tr>
    <td bgcolor="red">1</td>
```

```
<td style="background-color:#ff0000;">2</td>
<td style="font-size:2em;">3</td>
</tr>
</table>
```

В данном случае с помощью атрибута border я указал толщину линии границы внутренней и внешней. А затем с помощью bgcolor указал заливку фона ячейки красным, и то же самое, но с использованием атрибута style и прописанными стилями, проделал со второй ячейкой.

В случае с третьей ячейкой, я увеличил ей размер шрифта в два раза. С помощью относительных величин em.

Примеры верстки тегами

Простая страница с текстом

```
<!DOCTYPE html>
<html lang="ru">
  <head>
    <meta charset="UTF-8">
    <title>Document</title>
  </head>
  <body>
    <article>
      <h1>Название статьи</h1>
      <p>Lorem ipsum dolor sit amet, consectetur adipiscing
elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque
penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec
quam felis, ultricies nec, pellentesque eu, pretium quis, sem.</p>
    </article>
    <aside>
      <nav>
        <ul>
          <li><a href="#">Ссылка</a></li>
          <li><a href="#">Ссылка</a></li>
          <li><a href="#">Ссылка</a></li>
          <li><a href="#">Ссылка</a></li>
          <li><a href="#">Ссылка</a></li>
        </ul>
      </nav>
    </aside>
  </body>
</html>
```

Пример статьи с навигацией

В данном примере отсутствуют основные элементы страницы, как класс. А именно:

Шапка сайта (Header)

Подвал (Footer)

Тем не менее, с такой структурой можно вывести отдельно стоящую публикацию, например, статью и использовать для нее внутри aside дополнительную навигацию. Для сравнения можно привести аналогию с книгой. В самом начале идет заголовок, после него дополнительная многоуровневая навигация по этой книге, затем сама книга.

Навигация

```
<nav>
  <ul>
    <li>
      <a href="#">1 уровень</a>
      <ul>
        <li><a href="#">2 уровень</a></li>
        <li><a href="#">2 уровень</a></li>
      </ul>
    </li>
    <li>
      <a href="#">1 уровень</a>
      <ul>
        <li><a href="#">2 уровень</a></li>
        <li><a href="#">2 уровень</a></li>
      </ul>
    </li>
    <li>
      <a href="#">1 уровень</a>
      <ul>
        <li><a href="#">2 уровень</a></li>
        <li>
          <a href="#">2 уровень</a>
          <ul>
            <li><a href="#">3 уровень</a></li>
            <li><a href="#">3 уровень</a></li>
          </ul>
        </li>
      </ul>
    </li>
  </ul>
</nav>
```

```
</ul>  
</li>  
</ul>  
</li>  
</ul>  
</nav>
```

Пример навигации отдельным фрагментом

Данный пример отображает принцип сбора многоуровневого меню. Такой подход определенно имеет смысл применять для построения навигации в шапке сайта. Чтобы вывести разделы и подразделы, например.

CSS

Каскадные стили (CSS)

Апогеем в визуальном оформлении страницы являются каскадные стили. Слово каскадные в данном случае означает последовательность слоев применения стилей. Можно выразить и проще, сказать о приоритетности, но каскад тут, пожалуй, уместнее.

Происходит следующее: базовые стили тегов имеют самый низкий приоритет в визуальном оформлении (не путать с значением атрибута style), затем то, что указано в начале кода стилей и читается сверху вниз.

```
p {  
  color: black;  
  color: red;  
}
```

Пример каскадности в стилях

Попробуйте угадать, каким цветом в итоге будет окрашен текст внутри тега p. Он станет красным. Именно таким образом, посредством каскадирования решается вопрос конфликтов стилей.

Наследование

Вторым свойством стилей является наследование. Оно добавляет некой прозрачности в оформлении, только в отличие от каскадирования работает не так очевидно.

Например у вас есть блок `div`, внутри которого с помощью тега `p` выведен некий текст. Для дива прописано свойство `color:red;`. Далее, если для тега `p` отдельно не указаны иные свойства, оно примет это свойство родителя, т.е. дива и цвет текста станет красным.

Как использовать стили

Существует несколько способов интеграции каскадных стилей в верстку.

Прямо в тег в атрибут *style*.

Непосредственно внутри тега *<style>*

В отдельном файле с расширением *.css подключив его к html с помощью тега *<link>*

Первый способ, инлайновый, позволяет управлять элементом прямо в коде, прямо в описании этого элемента. Минус такого подхода в том, что в нем невозможно описать медиа запросы. Следовательно, вы сможете написать правила отображения только для всех экранов сразу.

Обращаю внимание, что при использовании javascript вы столкнетесь с тем, что с его помощью указываются стили для конкретного случая.

Тег *<style>* отлично используете для частных случаев, когда у вас есть общий сборник стилей в отдельном файле. Я часто применяю такую схему, когда необходимо провести пробу стилей для какой-то отдельной категории сайта, тогда я пишу в шаблон раздела все необходимые стили.

Еще хорошим примером использования тега *<style>* может стать индивидуальное описание конкретно взятой страницы.

Есть мнение:

Что стоит использовать исключительно подключаемый внешний файл. К сожалению, в верстке много таких «суеверий». Не стоит поддаваться им, следует подходить к вопросу критично и рационально. Т.е. исходя из условий и поставленной задачи.

Java Script

Джаваскрипт, он же EcmaScript был создан и продвигался, как мощный язык для работы с DOM. Но на заре его существования он представлял из себя разрозненные методы, у которых не было, да и сейчас почти нет, системы.

Никто никогда не мог понять, почему в сущности он стал таким популярным.

«Самый неправильно понятый язык программирования в мире стал самым популярным в мире языком программирования»

– Дуглас Крокфорд

Еще одной популярной технологией JS является AJAX – методы асинхронной передачи данных для создания интерактивного веб-приложения. Служит в основном для того, чтобы передать данные из сессии браузера в программную часть, например в скрипты PHP. И для получения и обработки данных, например в формате JSON.

После принятия стандарта ES6 с ним [JS] стало действительно комфортно работать. К текущим минусам можно отнести только нестабильную документацию и возможности. Что-то добавляется, что-то удаляется. Стабильно работает только некоторая база. Вторым пунктом идет тот факт, что одно и то же можно выполнить разными способами.

Но для изучения базовых возможностей и для «рядовой» работы хватит с головой.

Важно также отметить то, что это ООП – объектно ориентированное программирование. ООП – это методология программирования, которая базируется на создании объектов для управления. В данном случае мы имеем дело с объектами, которые несут в себе информацию для вывода содержимого на страницу [контента], настройки, конфигурации и т.д.

В веб-разработке вы скорее всего столкнетесь и с другим проявлением JS – node.js.

Адаптивная верстка

Адаптивная верстка

Не так давно, в 2010 году появился термин отзывчивого дизайна, который как раз предполагал попытку создать такой дизайн, который будет хорошо смотреться на всех устройствах.

Небольшой экскурс в историю

Когда термин только начал появляться в рунете, далеко не все, включая меня, понимали, что это такое и что с этим делать. Можно было наблюдать такие явления, как указание размеров в процентах. Или указание ширины дивов в точных размерах в надежде, что они правильно перенесутся на новую строку.

Медиазапросы

Перед тем, как продолжить рассматривать тему адаптива, следует отметить, что адаптивная верстка это не только про разные размеры экрана. Это также о типах выводимой поверхности: экран, печать на листе, устройство для передачи информации на базе шрифта Брайля и скринридеры.

Экраны в свою очередь делятся на наладонники, экран компьютера, телевизора и проектор.

Для каждого устройства свойственны свои особенности восприятия и передачи информации. Например для медиазапроса типа:

@media print

Стоит использовать измерения в саниметрах или процентах. Либо верстать с учетом того, что вы точно понимаете, в чем разница между печатью 72 dpi и 300 dpi.

Для запроса же типа:

@media aural

Возможно стоит учитывать тот факт, что пользователь скрин ридера вряд ли оценит иконочный шрифт или картинку. Или кнопку, в которой будет что-то такое

```
<button><i class="fa fa-angle-right"></i></button>
```

Так как пользователи скринридера «видят» сквозь озвучивание контекста, то в конкретно данном примере они «увидят» ничего.

Возможно вы встречали в том же Bootstrap класс sr-only. И с его использованием данная же конструкция станет выглядеть так:

```
<button>  
<i class="fa fa-angle-right">  
<span class="sr-only">Следующий слайд</span>  
</button>
```

Именно таким способом добавляя текстовую информацию для скринридера можно добиться читаемости контента.

Возможно,

Вы никогда и не встретите случай, когда потребуется сверстать непосредственно для скринридера или шрифта Брайля. Ибо вроде бы как бы это удел госструктур. Не расстраивайтесь, просто вспомните, что есть поисковые роботы, которые в данном случае подобны

слабовидящим и слепым и осуществляют навигацию примерно таким же образом. Не видя текста, он не поймет, о чем картинка, иконка.

Помимо определения типа поверхности директива `@media` позволяет определить и дополнительные свойства. Такие как `pixel rate`. Что в свою очередь дает возможность написать стили для экранов с повышенной плотностью пикселя.

И пишутся они, как дополнение через оператор И (and) . Например следующим образом:

```
@media screen and (max-width: 600px){  
...  
}
```

Таким образом мы проверяем, что стили, которые написаны далее в скобках будут применены строго в условии, если отображаемым устройством будет экран размер которого строго ограничен 600 пикселями.

Именно таким образом и проверяется размер этой самой поверхности. Но, помимо `max-width` и `min-width` существует еще вагон и маленькая тележка для проверок. Среди них:

min- max- width – Определяет ширину рабочей области, экрана в px, em; листа в cm.

Можно задать ограничение от, префиксом `min-` и до – `max-`

```
@media screen and (max-width: 600px)
```

min- max- height – В точности копирует ширину и накладывается на высоту области.

orientation – Благодаря этому свойству можно задавать разные стили для разной ориентации экрана. Наиболее актуально для мобильных телефонов, например, при горизонтальной (landscape) ориентации прятать плавающую навигацию, которая ощутимо откусывает место определенное под контент.

Может иметь 2 параметра:

Landscape

Portrait

```
@media screen and (orientation: landscape)
```

min- max- aspect-ratio – Определяет соотношение экранов. Например, вы верстаете админку с увесистой левой панелью, которая отлично смотрится на современных мониторах. И место для рабочей области есть и панель на месте. Но что делать, если с вашим веб-

приложением работают на экране с соотношением 4:3. Как вариант, можно скрыть панель при достижении классической ширины для таких мониторов в 1024px, но не исключено, что у данного товарища вполне большой экран с увесистым разрешением.

В таком случае имеет смысл проверить разрешение этим свойством:

@media (min-aspect-ratio: 4/3)

Обратите внимание на то, что запись производится дробью.

min- max- resolution/device-pixel-ratio — Если же, как я указывал выше, надо сделать разные стили для обычного экрана и для экрана с повышенной плотностью, вы можете использовать проверку этими свойствами. Поговорим о них чуть ниже.

За ретину и плотность пикселей

Говоря о проверке на плотность пикселя, не стоит забывать о том, что же оно в сущности из себя представляет.

Если вы возьмете в руки телефон с экраном 480x640, вам будет комфортно с ним работать. К тому же он будет определенно меньше жрать батарею, чем такой же, но с повышенной плотностью. Под такой же, я имею в виду физические размеры экрана. Например, оттолкнемся более предметно: iPhone 8 имеет физические размеры 138,4 мм. на 67,3 мм. И давайте представим, что у него экран не ретина, а обычный и имеет размер: 667 пикселей в высоту и 375 в ширину. С таким разрешением вы свободно можете смотреть фильмы, читать текст с экрана. Действительно. И в принципе размеры экрана, примерно с ладонь, укладываются в соотношение с пиксельной сеткой.

Но вернемся к действительности и откроем спецификацию. Размер экрана в пикселях: 1334x750. Казалось бы размеры в 2 раза больше. Но для сравнения откройте параметры экрана своего персонального компьютера. В большинстве случаев вы получите картину в разбеге от 1440 пикселей в ширину и до 1920. А теперь снова посмотрите на высоту iPhone 8.

1334 пикселя, которые помещаются в ладонь против 1440+, которые ну никак не поместятся (речь об экране).

Таким образом мы хотим получить более четкую картинку. К сожалению тут больше проблем, чем радости. Поясню на предмете обычного черного квадрата.

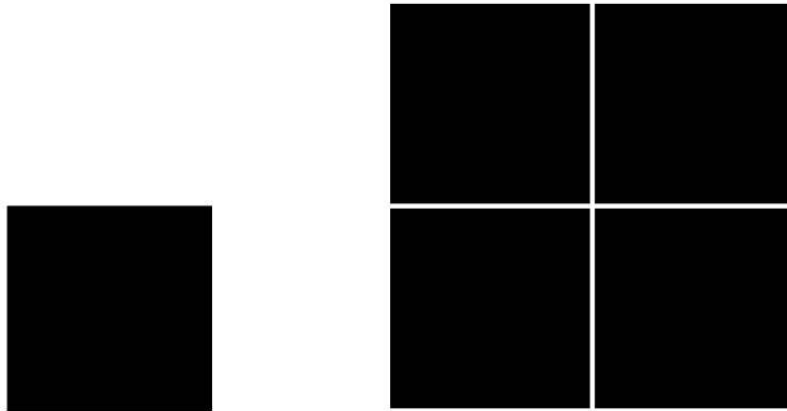


Слева мы имеем квадрат в 100 пикселей, поверьте на слово. А справа в 2 раза больше, т.е. 200 пикселей. И тут я сразу разочарую. Тут нет и не будет понятия в 2 раза больше. Так как когда мы говорим про разрешение экрана, речь идет о количестве пикселей, которые могут быть на нем размещены.

Иными словами экран iPhone 8 может передать нам 1 000 500 пикселей. В то время, как воображаемый iPhone с разрешением в 667x375 сможет передать всего 250 125 пикселей. Иными словами разница составит 4 раза.

Повлияет ли это как-то на отображения квадрата? Если вы правильно его экспортировали то нет, никак не повлияет. И ухищряться на предмет увеличения размера и, соответственно, веса не стоит. Маленький квадрат не размост от того, что его попытается воспроизвести экран с большой плотностью. И большой квадрат, соответственно, тоже никак не должен видоизмениться на маленькой плотности.

Итак, когда мы говорим о двойной кратности в разговоре о ретине!? Именно в тот момент, когда описываем факт, что вместо 1-го пиксела наш экран с повышенной плотностью этих самых пикселей может выдать в 2 раза больше информации в ширину и в высоту. Примерно следующим образом:



Пример сравнения пикселей с повышенной плотностью.

Как вы можете видеть, у нас и слева и справа по 1-му пикселу. Вот только правый может отобразить информации, как 4 таких, как слева.

Говоря отобразить информацию, я говорю о растровой графике. Где каждый пиксел хранит информацию о цвете, координатах и т.д. В то же время используя SVG, который является векторной графикой, формально можно пропустить эти условности по ретина\не ретина. Так как там задаются точки в пространстве, векторы.

Итак, как же все-таки дать CSS исключительно для экранов с повышенной плотностью.

```
@media only screen and ( min-device-pixel-ratio: 2),  
only screen and ( min-resolution: 192dpi){  
}
```

Пример отлова ретины

Таким образом можно проверить device-pixel-ratio, и разрешение в dpi (dots per inch). Они высвечивают диапазон для экранов с повышенной плотностью. К слову, если бы не стояло правило only-screen, то min-resolution применялось бы и для печати, так как изначально этот термин принадлежал ей.

К слову, о dpi. Изначально для дизайна экранов используется конфигурация в 72 точки на дюйм. Это стандартное «качество» для монитора. «Стандартным» качеством же для печати является 300 dpi.

Слово стандартным в кавычках, так как беря в пример выше описанные 2 квадрата, вы спокойно можете принести в печать тот самый квадрат экспортировать его в стандартном для экранов разрешении в 72 dpi. И несмотря на очевидное ворчание печатника, все-таки отправить в печать этот несчастный квадрат.

Семантическая верстка

Семантическая верстка

После освоения адаптивной верстки стоит прибегнуть к изучению, как верстать семантически корректно. Что же это значит!?

Данный подход означает использование тегов и атрибутов в соответствии с их предназначением.

Можно сравнить с языком – человека лучше понять и приятнее читать, когда его речь насыщена, грамотна и ощутимо легче понять, если он называет вещи своими именами.

В действительно мы сильно грешим некорректным использованием языка. Возможно вы слышали, как многие профессии связанными с IT часто называют программистами. Уже давно в ходу глумливая версия – ТЫЖПрограммист. Я уверен, что это отличная аналогия, близкая читателю. Или станет близкой после освоения профессии верстальщика.

То же самое происходит и с версткой страниц. Важной причиной тому тот факт, что разнообразие появилось в процессе развития языка разметки. Сперва у нас были таблицы, затем блоки, а вместе с приходом стандарта HTML5 появилось и продолжает расти количество тегов, разнообразие которых заставляет сперва запутать верстальщика. Ведь так просто продолжать использовать универсальный тег `div` для блоков чего-либо, и ограничиться простым тегом `p` для оформления текста. Не стоит поддаваться соблазну. Но для чего вообще это нужно!?

Для чего нужна семантическая верстка

В первую очередь – для удобства восприятия текста человеком. Я говорю про такие теги, как `cite`, `blockquote`. Их визуально можно отличить на странице, а если добавить стили, то это становится еще проще. Кроме визуального оформления оно еще и корректно воспринимается роботами, которые проходятся по верстке.

Роботы, поисковые пауки – вторая причина, для которой это необходимо. Чтобы ваша верстка корректно читалась и воспринималась роботами необходимо использовать такие теги, как `header` для определения шапки сайта, `footer` для определения нижней части сайта. То же самое относится и к отдельно взятым секциям на странице.

Для таблиц же есть `thead` и `tfoot`.

Существуют алгоритмы поисковых роботов. Немного информации о том, как работает поиск Яндекса можно подчеркнуть на их странице о технологиях.

Когда речь заходит о семантической верстке можно с уверенностью воспринимать это в качестве попытки способствовать чтению компьютером сверстанной страницы.

Больше информации в OpenGraph

Вдобавок в информации о семантической верстке тегами можно начать говорить про микроразметку. Если говорить очень грубо, то это верстка в верстке. Второй слой доступности для роботов.

Простой пример результата применения микроразметки – появление сниппетов в поиске. Дополнительной, вспомогательной информации, которая явно выделяет сайт в поиске.

Методологии верстки

Методологии верстки

Когда-то, и я это время застал, мы верстали так, как придется. Бессистемная верстка – название классов как латиницей, так и транслитом из кириллицы, отсутствие подхода к логике, отсутствие структуры.

Так верстают и сейчас, к моему сожалению, и часто приходится это наблюдать. В чем же кейс проблемы? Когда нет структурированного подхода, такие проекты сложно поддерживать. Они делаются на один раз, быстро. А как известно, то что делается быстро и на один раз – не стоит дорого, может быть вообще не стоит своих денег.

Не могу сказать точно, как мы пришли к некой логике и структуры, я лично сам много барахтался и пытался собрать свою систему, что-то получалось, что-то нет. И тем не менее, на данный момент у нас есть множество методологий верстки, это:

1. Атомарники;
2. БЭМ;
3. SMACSS;
4. AMCSS;
5. И многие другие, которые я никогда не использовал.

Возможно мне повезло, но так вышло, что моя система как минимум стремилась к изолированности БЭМ и классам помощникам в духе атомарника, поэтому, как только я на них подсел, больше не вижу смысла двигаться. И тут есть о чем поспорить и улучшить в своем подходе.

Атомарный подход

Поговорим про атомарный подход: он крайне прост, на каждое свойство CSS есть свой класс. Скорее всего есть классы и под систему медиа-запросов. Несколько кривя душой можно сказать, что Bootstrap-фреймворк это атомарник.

У подхода первым делом при знакомстве к крохам информации в интернетах появляется одна большая претензия – свойств CSS-то много, а вариаций дополнительных классов типа ``.color-blue`` может быть столько, сколько оттенков в проекте.

Например в том же бутстрапе есть большая сетка градаций серого, которые вы в проекте использовать скорее всего не будете. Переменные там выглядят, как ``$gray-100``, ``$gray-200`` и т.д.

А сам код выглядит не менее странно, чтобы задать параметры непосредственно тегу, порой возникают жуткие конструкции в духе:

```
...  
<p class="color-gray-100 font-family-arial line-height-1-66 font-size-18  
display-flex align-items-center ..." >...</p>  
...
```

При этом в моем примере все они человекопонятные. Из-за такой плотности и разнообразия их часто сокращают до, скажем, ``fs-18``. Что не позволяет понять без подготовки и подглядывания в документацию, о чем идет речь.

При этом я не утверждаю, что данный подход плохой. Но о выводах после БЭМ.

БЭМ

БЭМ (Блок Элемент Модификатор)

В идеале, я рассчитываю, что к этому моменту ты уже успел познакомиться хотя бы с термином, а то и как-то разобраться в «официальной документации», которая есть у наших любимых разработчиков Яндекса. Потому что дальше пойдет сложновато.

Итак, БЭМ расшифровывается очень просто:

1. Блок;
2. Элемент;
3. Модификатор.

Блок – по своей сути закрытый контейнер. Он ничего не знает про то, что вокруг. В себе же в это время он хранит набор (один и больше) элементов. По сути задача блока – описать своей внешний вид, поведение и поведение непосредственно дочерних элементов.

Элемент – можно сказать мини-блок, который находится непосредственно внутри блока. У него есть какие-то свойства и его родитель, блок, точно знает, как этот элемент должен себя вести. Естественно, элемент ничего не может знать про то, что находится снаружи родительского блока.

Модификатор – волшебная палочка как для блока, так и для элемента. Многие ошибочно полагают, что модификатор позволяет дать возможность блоку как-то узнать о происходящем снаружи, например задать `margin`, но нет, это [задавание отступов модификаторами] работает только для элементов внутри блока.

Примесь, микс – подход, который позволяет как просто вкладывать блоки в блоки в качестве элементов, так и полностью смешивать их структуру. Это сложно (на первых взгляд) и на примерах я разберу это ниже. Скорее всего ты сюда попал, разве не въехал с официальной документации, что такое примеси. Чтож, тут есть разглагольствование на эту тему.

Простейшим примером классического блока может быть логотип на сайте:

1. Есть сам логотип, он же блок, т.е. родитель.

- a. Внутри есть элемент картинка.
- b. А может быть и текстовая подпись.

Логотип может быть адаптивным, т.е. будет изменяться картинка, положение текста в зависимости от разрешения экрана. Или от текущего класса модификатора.

Например в шапке сайта надо использовать сразу маленькую версию изображения (иконку), а в подвале сайте – обычный полноценный логотип. При этом базовое поведение у них будет одинаково. Должно быть.

Это решает созданием одного блока и подгонки его под разные ситуации посредством модификаторов. Подходов может быть несколько и я их также разберу ниже.

Как разрабатывать по БЭМ-методологии

В то, что я описал выше вроде бы легко въехать, но на деле новички часто путаются на первых шагах и их часто бывает трудно преодолеть без наставника.

В любом случае, будь я в роли ментора или вот так, человек со стороны, первое, что необходимо сделать, когда начинаешь вникать в БЭМ – обратиться к генераторам древо образных структур в одностороннем порядке. Неплохим примером может быть MindMeister, другие генераторы блок-схем или ручка с листиком.

Последние два варианта в отличие от ментальных карт хуже тем, что можно создавать двусторонние связи, но если очень постараться и держать себя в руках, то этот подход ускорит понятие методологии.

Что же надо сделать – разработать структуру блоков в проекте. По отдельности и потом можно собирать вместе, словно конструктор.

Структурная часть

Возьмем за образец какой-нибудь типовой воображаемый прототип сайта. В нем будет:

1. Шапка;
2. Навигация;
3. Логотип;
4. Основное тело страницы;
5. Подвал;
6. Список новостей;

7. Новость;
8. Изображение;
9. Заголовок;
10. Параграф;
11. Ссылка;
12. Иконка;
13. И т.д.

Вот, что примерно должен разглядеть верстальщик в макете. И дальше остается руководствоваться логикой, например:

– Логотип в подвале и шапке выполняет роль ссылки, стоит ли вкладывать его в блок ссылки?

– Вряд ли, это вполне самостоятельный блок.

Внутри логотипа же могут и в принципе должны находиться элементы. При этом не обязательно вписывать на странице тот элемент, который не нужен.

Например в «мобильной» версии логотипа отсутствует текст, а в обычной он есть. Подстрочник, который можно выделить и скопировать. Да, пример граничит с безумием, но все же он простой для понимания, поэтому мусолим дальше.

Для того, чтобы скрыть элемент в «мобильной версии» логотипа вовсе не обязательно ему указывать `display:none`, можно просто не добавлять его в верстку\рендерить (на уровне JS).

Таким образом легко решается проблема переопределения стилей. Ведь если они не заданы, то и переопределять нечего и нет никакого смысла это делать.

Необходимо еще отметить, что описывать на уровне CSS структуру стоит в один уровень вложенности, даже если на деле в html вложенность глубже и не имеет жестких рамок. Если ты жестко связал структуру CSS со структурой HTML – выстрелил себе в ногу.

Модификаторы

Но для чего же тогда нужны модификаторы БЭМ? В принципе для всего другого. На примере того же несчастного логотипа мы можем определить в стилях изображение, которое необходимо использовать для модификатора мобильной версии логотипа.

При этом модификаторы можно применять как к блоку, так и к элементам внутри него. Более того, можно создавать сложные условия

и взаимодействия, главное при этом – сохранить доступность к пониманию, что тут происходит.

Более уместным примером будет, скажем, увеличенное изображение в логотипе (описанное в гайдлайне, конечно же) для заставки на прелоадере (не пишите мне, если используете прелоадеры, для вас существует отдельный котел в аду). Тут уже следует подумать. Увеличиться должно только изображение, или картинка и подстрочник. В первом случае это определенно модификатор элемента, а во втором это может быть модификатор блока, который будет пропорционально увеличивать как размер шрифта подстрочника, так и изображение.

Миксы по БЭМ

Самое страшное, что есть в БЭМ – миксы. Просто потому что официальная документация как-то очень вскользь по ним пробегается, хотя в реальной жизни миксовать приходится на каждый пук.

Далеко ходить не надо, обратимся к структуре выше и вспомним про шапку и подвал. Шапка и подвал – определенно блоки. А вот как в них правильно вложить логотипа и навигацию? Напоминаю, что блок не знает своего положения в мире, он описывает только внутрь. Поэтому отступы в шапке для логотипа определяет шапка. Как?

Все просто, у нас есть блока шапка, скажем `.header` (воу конкретные примеры пошли). И внутри нее есть элемент `.header__logo` – что этот элемент делает. Ну во-первых он миксуется с недавно созданным логотипом. Должно получиться что-то в духе:

```
.header  
→ .header__logo.logo  
→ → .logo__image  
→ → .logo__subscription
```

Итак, на втором уровне вложенности у нас появился микс. `header__logo` миксуется с `logo`. Зачем?

Все просто, лого, как я уже не раз говорил, не знает своего положения в обществе, оно описывает то, как должно работать лого и только. А вот элемент `header__logo` уже знает, как должно вести себя лого внутри `header`. Опасный момент, знает только как вести себя внутри `header`! Это важно.

По сути на этом моменте в замиксовке у тега с классом `logo` появляться новые отступы, может быть, и в принципе все. Переопределять свойства `logo` в миксе не стоит, ведь не стоит же

усложнять себе жизнь, верно? Почти, но это я тоже хочу разобрать в отдельном кейсе.

Тут-то осталась одна неразгаданная тайна, как `header` поймает свое положение на странице, если оно вроде как ничего не знает о том, что вокруг?

Все просто, в документе должен появиться блок страниц, например `.page`. Он станет корневым и с помощью элемента `.page__header` и микса с шапкой расположит ее в ее законном месте. Может быть, да? Не совсем, ведь есть еще модификаторы, которые позволяют в произвольном порядке модифицировать как блоки, так и элементы. Но как это применимо в данном случае?

Вот несколько простых примеров:

1. Обычный `page` содержит в себе статическую шапку и вроде бы как ничего не описывает для нее. Допустим только отступ вниз, чтобы основной контент не был прибит прямо к шапке.

2. Или вот у нас идет `heroscreen` весь из себя такой красивый и надо при прокрутке, чтобы шапка вставала сверху и крутилась с пользователем. Тут хватит добавить `position:sticky` шапке, а сделать это можно через модификатор элемента `page__header-sticky`.

Сам `page` скорее всего не будет иметь модификаторов, хотя иной раз надо задавать, например темную тему для сайта, а это можно как раз сделать одним махом для всей основной структуры задав модификатор `page-dark`. Хороший ли это подход? На самом деле не очень, так как все равно надо задавать подобный модификатор вложенным блокам, чтобы корректно их отрисовать. Переопределять цвет логотипа (svg например со свойством `fill`) через наследование в несколько миксов – плохая идея, специфичность не вывезет. Оптимальное спустить команду вниз, линейно, как в дереве и заложить это на уровне JS. Можно хоть `mutationObserver` прикрутить, если совсем лень-матушка.

Как называть классы БЭМ

Основной критерий, который очень важен – читаемость. Когда ты смотришь на HTML-код, то должен понимать, что будет происходить с содержимым, т.е. самой версткой.

Да, такой подход сильно удлиняет размер класса и как результат кода. Впрочем, есть смысл задуматься, а нужно ли такое усложнение вообще:

Размер можно указывать, как `-2x`, а не `-double-size`.

Цвет color можно опускать и задавать –white. Правда в случае с фоном придется добавить –background-white модификатор.

Стили\схемы наименований

Есть несколько схем наименований по БЭМ, я прошелся по нескольких и в конечном счете перешел на международный стиль. Когда элемент отделяется от блока двумя нижними подчеркиваниями, а модификатор двумя дефисами. Это выглядит вполне читаемо: block__element–modifier.

Конкретные примеры по БЭМ

Блок БЭМ с модификаторами, но без элементов

Очень просто – кнопка. По умолчанию скорее всего у вас будет просто `TextNode` в виде текста кнопки, зато в наличии будет пачка модификаторов для разных цветов, бордеров, начертаний.

Но это вовсе не значит, что в будущем не появится элемент у данного блока. Ничего не мешает появиться иконке, которое потребуется указать положение внутри кнопки.

По сути весь БЭМ – это задел на будущее. Создавая блок можно не описывать стили самого блока, зато задать условия для элементов, или наоборот не добавлять пока несуществующих элементов, сделав это позже.

Как сочетается с атомарным подходом

Кнопка, в которой есть только блок с модификаторами очень похожа на атомарный подход. Но все-таки это разные вещи.

Атомарный подход пропагандирует класс для всего. Точнее классы для всего, на каждое свойство и значение CSS будет свой класс, нужен цвет бирюзовый – будет класс `.color-turquoise`. Нужен отступ сверху в 23 пиксела – будет класс `margin-top-23px`.

Т.е. атомарный подход не требует наличия базового класса для блока, в отличие от БЭМ. В атомарном подходе каждый класс самодостаточен, в БЭМ это обычно база + модификатор, который приносит дополнительное описание.

Создавать обертку (wrapper) или нет?

Классика, быть или не быть, создавать ли для `news` класс-обертку `news-wrapper` или нет?

Конечно же нет, нормальный подход наименования классов гласит – названия классов должны отвечать на вопрос, что он делает. А `wrapper`, обертка, что?

Рассмотрим частый кейс – есть карточка новостей. Как описать поведение группы карточек?

Допустим для класса карточки новостей мы создали класс `news-card`. Внутри будут элементы, но они нас не волнуют. По первому пути, тривиальному, у нас будет просто список карточек. Его спокойно

можно обозвать news-cards-list. И модификаторами можно задавать разное количество карточек в строке.

Немного усложним, и представим, что в новостях надо представить, как список карточек, так и элементы навигации, например параметры сортировки. Просто так в список им будет трудно войти, поэтому у нас есть опять же, несколько путей:

1. Создать блок news, внутри которого будет news__nav и news__list, а еще можно протянуть и news__item.news-card в такой вот микс, да.

2. Создать блок news, все как выше, но не вкладывать напрямую, а сделать блок-прослойку – news-list, у которого будут элементы news-list__item.news-card.

И второй вариант будет предпочтительнее, так как позволит не уходить слишком глубоко (фактически в HTML) в плоской структуре стилей. Но в конечном счете все зависит от макета. Так как news-card может быть использован и вне списка новостей. Может быть это будет отдельно стоящая карточка с новостью в тексте статьи. Помним, блок описывает только то, что знает наверняка – себя и свое содержимое, поэтому с легкостью может быть встроен почти куда угодно, ведь внешние положение за него опишет элемент родительского блока.

Классу active быть или нет

Я часто повторяю, что при все своей красоте БЭМ – это вовсе не панацея и уж точно не свод жестких правил. Поэтому да, для каких-то случаев, где есть скрипт, может быть уже готовый, или просто очень лень писать всю, или может нужно написать один для всех элементов, вполне можно написать модификтор без полной записи. Т.е. например tab, на который нажали будет не tab.tab-active, а tab.active.

Хотя в конечном счете при адекватной сборке хватит обычной реактивности и триггера, которые должен отдавать true или false значение, чтобы активировать класс элемента, который был заранее указан. При этом наименование этой переменной в JS или ЯП бэкенда значение не имеет. В рамках разумного, конечно, читаемые названия это важно.

Файловая структура по БЭМ

То, что можно найти в документации на этот счет – вполне недурственный подход, который на самом деле в реальной жизни не

очень подходит.

Ключевая проблема – хранение файлов в одной директории. Это удобно, когда у тебя один инструмент сборки проекта, но совсем не годится, когда у тебя часть собирается по-одному, а часть иначе.

Хотя пакетные сборщики, конечно, могут спокойно и обойти собрать, что надо.

Как хранить файлы

У нас есть в среднем 3 типа файлов:

1. Стили;
2. JavaScript;
3. HTML

И вот с последним как раз проблема. Его нельзя собрать из чистого HTML в проект так, чтобы он принимал в себя значения, и уже даже при шаблонизаторе собирать разные структуры блока это некоторый геморрой.

А ведь в самом начале я писал, что если элемент не нужен, достаточно просто его не указывать в структуре блока. Это много лучше, чем скрывать DOM-узел.

Поэтому в большинстве случаев скорее всего в проекте не будет отдельно-стоящего HTML-файлика, разве что в качестве образца. В `jinja2` своя область видимости и совать в нее CSS и JavaScript не имеет никакого смысла. В проекте без шаблонизатора на PHP скорее всего у тебя будут свои какие-то уловки для генерации блока. К слову, миксовать на уровне бэкенда не так уж и просто.

Итого – да, хранить файлы отдельно каждые в своей папке с названием блока – удобно. Не менее удобно и заводить отдельную структуру для Sass-файлов, где все блоки будут лежать непосредственно в папке `sass/blocks/`, а js в `js/blocks/`.

Конкретные примеры исключений

Моей первой попыткой объединить все файлы в одной структуре работали в FA-Kit-е на базе языка `rug`. Но это даже не вышло в какой-либо релиз. Удобно собирать только темы для Themeforest. Статические и без ничего.

Апогеем же разработки стали для меня однофайловые компоненты Vue, где в одном `.vue` файле сразу html, js и стилей (притом без разницы на каком препроцессоре написанные). На самом деле там и на `rug` можно структуру оформлять.

А props-ы и расчетные data значения позволяют рендерить отдельные элементы или нет.

И тем не менее даже так подавляющая часть стилей у меня в проектах лежат отдельно, разве что я разбиваю их обычно на:

1. Blocks; 2. Pages; 3. Sections; 4. Layouts.

Все дело в том, что я активно использую препроцессор Sass и не каждый бэкенд умеет нормально с ним работать, а в качестве оптимизации я предпочитаю собирать базовые и уникальные стили для разных сценариев. Поэтому pages и layouts содержат эти условия.

А блоки, которые несут в себе какую-то уникальную информацию, или больше, чем какой-то блок, да еще с конкретной смысловой нагрузкой – такие я предпочитаю выносить в папку sections.

Это все позволяет держать в относительном порядке структуру проекта.

Что на счет реальной жизни?

На самом деле данная идеология не заканчивается на уровне классов. Навыки можно спокойно применять и дальше, формируя нормальные человекочитаемые переменные в JS или ЯП бэкенда, за которые не хочется надрать задницу автору переменных типа `lkCr`, `aPP` и т.д.

Но можно пойти еще дальше и структурировать директории примерно подобным образом. И называть файлы типа `footer-wave-small.svg` и `footer-wave-big.svg`. Тут сам БЭМ уже сильно с натяжкой, но ты должен понимать, к чему я клоню.

А что касается гибридного использования методологий, то в реальных проектах так обычно и получается, ты создаешь логику, которую легко поддерживать, а для случаев, скажем, класса `visually-hidden` будет использован атомарный подход, так как это не является самостоятельным блоком, скорее техническим классом, как я их обычно называю. Вспомогательным.

Оптимизация

Оптимизация

Оптимизация верстки и сайта в целом – это процессы, которые должны привести к тому, чтобы сайт меньше весил, быстрее загружался и быстрее работал.

Зачем это нужно, ведь есть 100-мегабитные скоростные соединения, что позволяют с большой скоростью загружать что угодно из интернета.

Все просто: все факторы о загрузке страниц сайта напрямую влияют на удобство пользователя и на место в поиске.

Во времена диалогов мы были готовы ждать, когда загрузится информация, неделями качали торренты, и отчаянно нервничали при сбросе соединения. И я могу сказать, что скорость развращает. В силу моей профессии и кочевого образа жизни, мне приходится использовать разное качество и скорость интернета.

Прямо сейчас, когда я пишу, я пользуюсь мобильным 4G интернетом от Yota. Сигнал перебивают стены и вышка Билайна, которая находится рядом, так что я испытываю трудности в загрузке страниц. Поэтому могу в полной мере оценить преимущества оптимизированных сайтов.

Что же такое техническая оптимизация сайта?

Основная проблема разработчика – чтобы пользователь получил необходимую информацию до того, как ему надоеет ждать. В большинстве случаев в этой гонке выигрывает тот, кто в поиске находится рядом и у кого страницы загружаются быстрее.

Если же у вас происходит очень длительная задержка, более 10 секунд, то с большой вероятностью пользователь уйдет. Ну, или, скажем, не дойдет.

И задача разработчика, который занимается оптимизацией сайта обеспечить корректную последовательность загрузки и оптимизировать размер зависимостей так, чтобы они и в качестве не потеряли, и весили меньше.

Условно загрузку можно разделить на несколько составляющих:

1. До первых байтов – время, что уходит на то, чтобы ваш запрос дошел до сервера и начал получать информацию в ответ. По сути мало от вас зависит, ну разве что конфигурация сервера, его ошибки, удаленное расположение и т.д.

2. До первой визуальной информации – появился текст, вуаля, пользователь начал его просматривать. Картинка? Еще лучше.

3. До начала взаимодействия – момент, когда пользователь не только увидел информацию, но и может листать страницу.

4. Время полной загрузки сайта.

Визуально в браузере ПК можно отследить по тому, что перестал крутиться индикатор загрузки на вкладке.

Алгоритм оптимизации сайта

Давайте начнем с вопроса оптимизации в том ключе, как это следует делать с точки зрения минимальных требований для запуска проекта. Такой подход предполагает:

1. Изменение размера изображений – сжатие и кодирования их в сжатом формате, например jpg2000 или progressive jpg. Сложнее с изображениями с прозрачностью.

2. Сжатие всех зависимостей кода – так называемый процесс minify\uglify, когда весь CSS или JS код из многострочного читаемого превращается в нечитаемый (практически) код в одну строку. Пробелы и переносы – это тоже символы. И они достаточно много весят в конечном счете. Просто сжав файл, удалив пробелы можно уменьшить его размер раза в два.

3. Перенос всех зависимостей в конец кода – т.е. сперва будет загружен контент в сыром виде, потом к нему применяются стили, затем JS код и так далее.

4. Сжатие конечного кода страницы HTML – сжатые зависимости это конечно хорошо, но еще лучше, когда все сжато. Такой код становится практически нечитаем пользователем, и в какой-то мере оберегает от копирования с целью использования своих данных. Выправить конечный код страницы в рабочий реально, но сложно, или, скорее, долго и нудно.

Следует отметить, что оптимизация происходит и в бекенде. Но все, что происходит в бекенде остается в бекенде.

Но на данных шагах рядовой разработчик остановится. Вам же выпал шанс узнать больше, в том числе получить несколько советов в определенных средах разработки.

И еще раз напомним, все что описано выше на тему оптимизации – обязательно входит в последующие шаги. Нельзя перепрыгнуть через сжатие изображений. Как бы ты ни старался, если картинки размером только выгруженных из памяти фотоаппарата – пиши пропало. Чтобы действительно сделать легковесный сайт, надо как минимум:

1. Включить gzip сжатие (страница упаковывается, передается и распаковывается на стороне клиента)

2. Включить кеширование ресурсов. Это позволит старым пользователям быстрее получать ресурсы из браузера. Старым пользователем считается после первого посещения с полной загрузкой кэшируемых ресурсов.

3. Интегрировать эти самые ресурсы в тело страницы.

Кэширование

Давайте немного остановимся и поговорим про данный аспект веб-разработки. В рамках вопроса оптимизации следует взять 2 вида:

1. Кэширование ресурсов на стороне клиента.
2. Кэширование ротируемого контента на стороне сервера.

Слово одно, а смыслы разные.

Кэширование ресурсов на стороне клиента – это сам документ, страница, что открыта у него в браузере, весь текст, ссылки на зависимости, такие как JS или CSS-файлы, или даже видео или треки.

При первой загрузке браузер пользователя все эти файлы старательно загружает, если может (404 никто не отменял, и игры с провайдерами тоже), а затем просто достает их из памяти компьютера. Таким образом нет необходимости ждать, когда все загрузится, что убивает скорость и качество интернет-соединения из факторов. Притом следует отметить, что все эти файлы могут быть загружены с первой посещенной страницы, например с главной. И при переходе на страницу, скажем блога, все загружаться уже не будет, а только то, что будет встречено новое, обычно это содержимое документа, т.е. текст страницы, картинки, ссылки.

Файлов в вебе, как известно, великое множество, каждый из них при загрузке формирует свой запрос и, если снизить их количество, то конечному пользователю будет удобнее и легче получить то, за чем он пришел на ваш сайт.

Второй же вид кэширования касается динамического контента. Описывая документ выше я затронул главную страницу и индексную блога. Обычно эти 2 страницы содержат вычисляемый контент бэкендом. Например для того, чтобы вывести информацию в виде списка ссылок, описания и изображения публикаций в блоге, движок обратится к базе данных, затем получит эти данные и отдаст во вьюху [view]. Конечно, если вы сами руками верстаете список записей и при обновлении переверстываете, то данные проблемы не про вас. У вас они другого характера.

Чтобы снизить время задержки ответа бэкенда, нагрузку на сервер такие виды (вьюхи) кэшируются. Точнее – следует кэшировать. Потому что представьте, вы заходите на сайт и в этот момент бэкенд

нагружает систему: оперативную память, процессор, хранилище и т.д. для того, чтобы получить из базы ответ и вывести его. А теперь умножьте это на количество пользователей, которые одновременно зашли посмотреть. Упереться в потолок и завалить сайт таким образом несложно, особенно, если движок не оптимизирован. Не техническим языком: движок сохраняет слепок готовой страницы, достав все данные из базы и отрендерив это один раз, и затем отдает на запросы сгенерированный файл.

Если соединить эти 2 метода вместе, а это в принципе нужно делать, то мы сокращаем время ожидания ответа сервера клиенту на запрос, нечего высчитывать, достаточно отдать уже готовое, и быстро подсовываем несохраненные зависимости.

В данной книге нет описания, как включить кэширование данных, это ищите в описании движков, которые вы используете, спецификации фреймворков, обсуждайте с бэкендщиками.

Отмечу только, из всех систем, с которыми мне приходилось работать, только 1С Bitrix имеет включенное по умолчанию кэширование данных, что, в свою очередь сбивает с толку, если об этом не знать. Ты можешь вносить изменения, долго ждать и пытаться понять, почему ничего не произошло.

Интеграция зависимостей

Интеграция зависимостей в тело страницы

В мире веб-разработки используется великое множество различных языков программирования, разметки и всевозможных фреймворков. В конечном счете они формируют один и тот же HTML-код с CSS и JS.

В каждом из них существуют свои нюансы, обусловленные возможностями языка. Проще всего в данном случае с PHP. Все, что нужно – заинклудить необходимую зависимость прямо в тело.

Вставка стилей в PHP

Чтобы реализовать это в PHP, необходимо использовать следующий код, к примеру:

```
<!-- Main CSS include -->
<style>
    <?php include 'style.css' ?>
</style>
<!-- Main CSS include END -->
```

Можно перестраховаться и использовать `include_once` для того, чтобы скрипт или стили не были вставлены еще раз.

Вставка стилей в Django

В данном случае существуют ограничения. По сути они упираются в возможности `jinja2`. А именно – ограниченность в видимой области при использовании с Django.

Сам `jinja`, надо отметить, имеет из коробки большой функционал, чем он же в Django.

Итак. Основная проблема в том, что нет возможности заинклудить все, что хочется. Следует использовать либо плагины, либо настроить сборку фронта так, чтобы все файлы, которые надо интегрировать, оказались там, где надо. В папках `template` приложений. Именно там находится зона видимости функции `include`. Например у нас основная

часть сайта выполняется приложением app. И для того, чтобы на главной странице подгружались файлы, выполняющие алгоритм masonry, мы используем такую конструкцию.

```
{% block addjs %}  
    {% include 'app/assets/js/script.js' %}  
    {% include 'app/assets/js/script.js' %}  
{% endblock %}
```

Для этого у нас настроена система распределения фронта по файлам, их копирования в конечные точки. О распределении и рецептам gulp немного позже.

Вставка кода в Jekyll

Мало чем отличается от Django, если честно. Весь ваш фронт должен быть собран в папке `_includes` для того, чтобы быть распознанным и инклудиться соответствующей функцией.

Почему код должен быть минифицирован

Кроме того аргумента, что код без пробелов меньше весит, его необходимо минифицировать для того, чтобы не возникало конфликтов. Алгоритмы минификации JS умеют минифицировать JS-скриптов, CSS стилей – CSS. Минификаторы для HTML далеко не всегда обладают возможностями с легкостью минифицировать вложенные зависимости. Именно поэтому они требуют предварительной обработки.

Все это не сложно, если для сборки фронта использовать сборщики проектов, такие как Gulp или Grunt.

Некогда я использовал Grunt, но затем перешел на Gulp в пользу простоты и расширяемости. Мнение сугубо субъективное, но рассматривать примеры мы будем на Gulp.

Распределение ресурсов зависимостей

JS зависимости

Особо критична данная тема для JavaScript, так как классы должны быть объявлены раньше, чем они будут использованы в коде. В отличие от функций.

Тем не менее такие библиотеки, как jQuery, если они используются, должны располагаться до того, как будут использованы зависимости построенные на возможностях данной библиотеки и ее классах.

Речь, конечно же, о популярном Bootstrap, который многие используют даже тогда, когда нет необходимости. И такие библиотеки, как masonry. И многие другие.

CSS зависимости

Другое дело обстоит с CSS файлами и кодом. Нет прямой зависимости в конечном виде кода за исключением самой сути стилей – каскадности. Это значит, что стили, объявленные ранее в равных условиях и, объявленные ближе к концу страницы, будут переназначены на более поздние. Это, опять же, отдельная тема.

В целом, бытует практика сбора всех стилей в один файл, которые затем просто «пристегиваются» к телу страницы в конце файла. Некоторые используют gulp-алгоритмы для того, чтобы вычистить неиспользуемые стили. Но в динамических сайтах с этим проблема. Никогда не знаешь наперед, что будет включено в сборку. Речь, конечно же, о пользовательском вводе в формате верстки. Ибо, если взять в пример, скажем, «Хабр Q&A», то там, что бы не ввели пользователи будет один и тот же набор элементов, стили для которых нужно подгрузить. Я имею в виду поля ввода, блоки вопросов и ответов на них, общие стили: шапки, сайдбара, подвала и т.д.

Такие темы в меньшей степени волнуют сайты, которыми занимаются полно укомплектованные команды, которые могут обеспечить такие изменения вручную.

И такую сборку можно также включить во вложенные зависимости. Но есть смысл создавать отдельные файлы для отдельных случаев, к примеру:

1. Шапку, подвал и основное оформление объединить в файл, к примеру `main.css`.
2. Туда же отнести типографику.
3. Использовать интеграцию кода для `carousel` для страниц, где есть `carousel` и подобным образом разделять остальные части стилей.

Закономерный вопрос: как определить, есть на странице карусель или нет? Варианта, пожалуй, два. Можно написать сложный алгоритм парсинга `html`-кода, сравнения его с каскадными стилями или просто руками добавить эту зависимость в шаблон. В первом случае есть более простой вариант сборки фронтенда – компонентами, например `Vue`, что позволяет собирать под компоненты отдельно стили и, грубо говоря, грузить их тогда, когда они нужны.

В большинстве же случаев я склоняюсь к последнему варианту. Опирируя шаблонами, я создаю островки контента, которые упорядочены некой визуальной иерархии, которую надо изменять не так часто.

Таким образом в общем и целом шаблоны остаются неизменны. А в основной файл со стилями следует добавлять только то, что будет использовано на каждой странице.

Такой подход актуален даже при использовании компонентной сборки с динамической подгрузкой стилей, так как позволяет даже так не засорять ненужным, не загружая в браузер пользователя неиспользуемые компоненты. В любом случае приходится искать баланс.

Как распределить эти зависимости

Как распределить эти зависимости

Все на удивление просто. Достаточно отобрать в соответствии с логикой загрузки важные части, такие как шапка сайта, и загрузить их сразу. Затем дать прогрузится контенту и выдать все остальное.

В некоторых случаях зависимости можно добавлять на лету для формирования частных случаев. К примеру, у меня на странице иконок в маркете использована интеграция стилей иконок. Для того, чтобы можно было отобразить весь список иконок.

В целом, я придерживаюсь следующей последовательности:

1. Начало загрузки документа `<head>`
2. Интеграция важных стилей, к примеру описание шапки
3. Загрузка контента с возможными интеграциями
4. Инклюд остатка стилей (основная масса с типографикой и каруселями)
5. Инклюд jQuery
6. Инклюд Popper.js (если дальше будет Bootstrap)
7. Инклюд Bootstrap-a
8. Инклюд кастомного Javascript
9. Загрузка внешних зависимостей, например FontAwesome
10. Инклюд калькуляторов счетчиков типа Google Analytics.

Практически все, что я описал выше, было подготовкой к пониманию данного алгоритма, но следует оговорить некоторые моменты.

Также не стоит забывать про то, что в момент загрузки страницы пользователь не обязательно окажется вверху тела страницы, а может быть где-то внизу и, порезанные неразумным образом стили, могут спровоцировать визуальное отторжение.

Почему мы откладываем загрузку JS как можно дальше

JavaScript требует для начала запуска полной загрузки документа. Т.е. находясь в конце файлы, для запуска все равно обычно ожидается

триггер в виде отслеживателя событий типа *window.onload => ()* или *document.addEventListener('DOMContentLoaded', function () {})*.

Вторая конструкция предпочтительнее, хотя и выглядит страшно. В отличие от *window.onload*, может срабатывать несколько раз в коде.

Почему мы откладываем загрузку счетчиков

Разумнее было бы загружать их в начале страницы. Чтобы они успевали фиксировать отказы, например. Но как показывает практика, в общем и целом с этим нет проблем.

Загрузка типографики

В рамках концепции «дай сперва увидеть содержимое», я сперва стараюсь отдавать, собственно, контент, а затем его оформлять.

В файлах стилей всегда есть перечень шрифтов, основных и «прикрывающих».

Я активно использую Google Fonts для привязки шрифтов, поэтому их я подключаю не инcludя. Дабы сохранить пути относительно файла стилей. Хотя это тоже возможно и реально..

Этот сервис предоставляет полезную функцию, вторую пару запасного шрифта:

font-family: 'Roboto', sans-serif;

Это значит, что в случае недозагрузки Roboto, будет использован другой доступный шрифт семейства sans-serif. А когда уже загружается необходимый шрифт, он «встает» на свое место.

То же касается и иконочных шрифтов. Если ваш сайт слишком тяжеловес при всей оптимизации, то даже откладывание на потом не спасет от видимых мутаций сайта в процессе загрузки. А если все сделано правильно, то отсутствие иконок на их местах на доли секунд не так критично хотя бы чисто визуально.

Рецепт Gulp для Django

Рецепт прост настолько, насколько сложно мне было до него дойти. В Django присутствует иерархия приложений. Т.е. каждое приложение отвечает за какую-то одну часть сайта (мы говорим о вебе, не забывайте).

На примере моего сайта есть приложение mainsite, в нем собирается фасад. Простые страницы по сути, в которые стекаются данные из других приложений, к примеру из кейсов. 3 последних кейса приходят сюда и отображаются на главной.

Все приложения равны друг-другу в плане иерархии директорий. Поэтому для gulp я собираю древообразную систему: от главного gulpfile.js до приложений. Подробнее об этом можно прочесть в источнике моего (<https://maxpoletaev.ru/blog/frontend-in-django/>) вдохновения.

К слову, спустя некоторое время frontserver у меня отвалился, а вот сборка gulp осталась. Единственный минус – на локалке запускается 2 процесса, один runserver окружения django и другой gulp watch для отслеживания изменений.

Мы отошли от рецепта по ссылке в том плане, что нам надо было складывать конечные минифицированные стили и скрипты в папке, в которой Jinja будет их видеть. Это папка templates внутри приложений.

Основная часть собирается у нас исходя из логики шаблона, который находится в папке mainsite/templates/layouts/index.html.

И все пути js-кода, и css ведут в папку mainsite/templates/assets/*, где распределяются посредством локальных gulpfile.js.

К моему сожалению, встроенный тег ssi удален, вероятно в связи с безопасностью. Он позволял инклудить с абсолютной ссылкой.

Такие манипуляции я произвожу даже со счетчиками. Они требуют небольшой доработки, но легко автоматически склеиваются.

Сложности доставляют библиотеки типа jQuery, Popper.js, Bootstrap, приходится их распахивать по отдельным тегам <script>.

Минификация изображений и отложенная загрузка

Минификация изображений и отложенная загрузка

Манипуляций с кодом недостаточно для быстрой загрузки страницы. Ведь основной вес – это медиа. Даже сжатые и оптимизированные медиа-файлы.

Речь идет об изображениях. Как я уже сказал, обычно считают, что хватает выбора оптимального размера либо с учетом перестройки для разных разрешений (медиазапросы), либо использование ресурсов для этих запросов:

```
<picture>  
  <source srcset="big-image.png" media="(min-width: 600px)">  
    
</picture>
```

И это работает. Действительно позволяет снизить нагрузку на мобильные телефоны и мобильную сеть. Я имею в виду медиа запросы, в которых на телефонах пользователь получает существенно меньшее изображение, чем получил бы на ПК.

Но таким образом сложно получить изображение высокого качества. Т.е. можно, но она будет тормозить страницу. Лучший способ – отложить загрузку. Данный метод я стал использовать не так давно, и сначала не использовал плейсхолдеры.

Алгоритм отложенной загрузки изображений предполагает:

Загружается документ

Собирается список изображений, что требуют загрузки

Загрузка по очереди

Выдача по факту загрузки изображения

В случае, если загрузка изображений запускается сразу на их места, по факту загрузки документа, выглядит это не очень хорошо. Хорошо, если это progressive jpg, он загружается поэтапной прорисовкой. Другое дело обычный jpg, который загружается сверху вниз посредством развертки.

Более оптимальным на наш взгляд является вариант, когда изображение на свое место попадает уже загруженное.

Это позволяет сделать наш скрипт ленивой загрузки изображений, который берет ссылки, создает экземпляры для загрузки изображений и после их загрузки в этом «вакууме» «переносит» их на место, заменяя placeholder.

Placeholder

Как я уже отметил, делать ленивую загрузку не подложив на место изображений что-то другое – глупо. А в случае с masonry выкладкой вообще не есть хорошо. Конечно, можно повозиться и сделать плавную загрузку карточек после загрузки изображений, но для этого необходимо регулировать поток загрузки, очередь подачи, иначе сортированные по дате посты с картинками превращаются в кашу.

Да и чисто эстетически приятнее смотреть, хоть и доли секунды, на вмняемые placeholder-ы.

Сев за проблему отображения картинок во время их загрузки, я сразу столкнулся с тем, что надо вместо изображений выдать что-то осмысленное.

Долго думал, подходил, рисовал, но пришел к простому решению: все, что выглядит, как общее – должно так и выглядеть. Т.е. просто прямоугольная форма с иконкой картинки. Иконку я, кстати, взял из моего набора иконок Rage.

Исключением из ряда стали плейсхолдеры для лиц. Для них я использовал иконку юзера.

Размеры изображений

Должны соответствовать размерам заменяемого. И это вызывает сложности. Так как выработать единую систему для форматов изображений бывает сложно. В случае с явным пользовательским вводом так вообще невозможно.

Впрочем, если быть честным, то возможно. За основу можно взять сохранение в базу данных изображений, и, взяв их размер формировать, создавать им связи с плейсхолдерами. Это увеличит нагрузку в момент загрузки пользователем картинки и во время получения данных из базы. Поэтому есть смысл решать этот вопрос, как все – задавать фиксированный размер картинки и давать пользователю имитацию возможности управлением им. Это даст возможность поместить, например, лицо в вырезаемую область, как при регистрации в социальной сети.

Тонкости

Как браузер «читает» код

Особенности чтения кода заключается в том, что это происходит сверху вниз по DOM модели. Говоря сверху вниз, имеется в виду в том числе вложенность.

Например, зайдя на сайт, браузер сперва оценит первую строку:

→ `<!DOCTYPE html>`

Оценит, по этой строке, что ему предстоит увидеть и продолжит.

Далее схематично пройдет, например так:

→ `<html>`

→ → `<head>`

→ → → `<title>`

→ → `<body>`

→ → → `<main>`

И так далее.

И это естественным образом влияет на передачу информации. Как минимум это [рендер страницы] занимает время. Зачастую рендеринг происходит прямо на глазах. Сперва, если мы не указали подключение стилей в шапке, мы увидим голый форматированный текст и изображения. Если же мы подключили файл со стилями сверху, то сперва браузер потратит время на то, чтобы прочитать этот файл. И потом примется за сам документ.

В этой связи существует множество проблем, которые упираются в специфику языка CSS и JS.

Проблема последнего – например тот факт, что классы должны быть объявлены раньше их использования. Вспоминаем библиотеку jQuery, которая из них состоит. Затем должны идти либо функции, либо код. Функции, к слову, могут быть и после их вызова в коде.

Итак, представим. Вам необходимо загрузить страницу и выполнить код JS. Например слайдер на главной у вас в чистом виде формируется в JS-коде. Если этот файл много весит, то при подключении его внутри тега head, браузер может потратить существенно много времени на то, чтобы его прочитать. Но тогда нет проблемы в рендере страницы.

Для обеспечения более быстрой загрузки, такие файлы нередко подключаются в конце документа. При этом для обеспечения синхронизации загрузки DOM модели и JS код используется триггер,

который ждет пока HTML будет загружен. Иначе, ваш прекрасный скрипт может запуститься до того, как будут отрендерены теги, в которых должен быть этот самый слайдер.

При этом же, чтобы обеспечить более быструю загрузку самой страницы, скрипты можно загружать с флагом `async`. Или `defer` (ждать загрузки документа).

Для стилей можно использовать конструкцию `preload`.

Обращаю внимание

В обиходе существует 2 крайности. Подключать файлы в `head` и в конце документа. Заявляю со всей ответственностью: подключить файлы стилей или скриптов вы можете где угодно. Главное помнить про рациональный и критический подходы к верстке.

Хаки

В силу того, что браузеры все еще по-разному смотрят на код, есть необходимость использовать хаки. Хаки в верстке – это специальные инструкции для конкретного браузера. Самым конкретным браузером всегда был Internet Explorer (IE) от компании Microsoft. Вторым по крупности стал их же Edge.

Благо не так много людей используют эти браузеры, поэтому порой можно их игнорировать. Тем не менее, никогда нельзя забывать про тот факт, что нужно четко понимать, кто является вашим пользователем. Вам может казаться, что к вам заходят только с Chrome, но стоит поддержать сайт пол года под избыточным трафиком и отслеживать браузеры с помощью метрик, как все станет на свои места.

Итак, чтобы описать стили именно для IE10–11 вы можете использовать, например, следующую конструкцию:

```
@media screen and (-ms-high-contrast: active), (-ms-high-contrast: none) {  
  /* стили только для IE10 IE11 */  
}
```

Так как я осознанно отказался от поддержки для старых браузеров, то не могу утверждать достоверность данной конструкции. Но считаю важным отметить такую возможность.

Грамотность

Появление компьютеров в нашей жизни сильно ударило по грамотности профессиональной прослойки населения. Хороший пример – использование дефиса вместо тире, неграмотное использование знаков препинания, не говоря уже о построении слов. Я сам не являюсь хорошим примером грамотности, так как часто забываю про те символы, которые не представлены в видимой части клавиатуры. Но немного терпения и когда-нибудь я сам себя приучу использовать alt набор и получать всегда корректные знаки там, где они необходимы.

Ревностными защитниками правильного набора текста, насколько мне известно, являются Артемий Лебедев и Артем Горбунов.

К частым ошибкам можно отнести:

Отбивка пробелом от символов, например часто вижу в переписках, когда используют пробел перед точкой или запятой.

Использование отбивки (пробела) слева и справа от дефиса.

Очень не рекомендую следовать легкому пути использования ошибок. Включайте почаще мозги и ваша ценность, как профессионала, будет расти.

Использование спецсимволов

Плавнo переходя из предыдущей секции поговорим про спецсимволы. Их существует великое множество и делятся они на нашу русскую и нерусскую.. Я сильно упрощаю для понимания русского человека, так как не «наши» можно разбить на еще большее количество видов.

К нашим можно отнести кавычки. Их использование разное вплоть до языков. Всех уравнила клавиатура, которой присутствует только один вид кавычек. Который в свою очередь достаточно неоднородно и хаотично интерпретируется в разных редакторах начиная от MS Word, заканчивая редакторами кода.

Тщательно выбирайте редактор кода (IDE), так как он может существенно испортить вам жизнь при наборе текста. Вторым же фактором является шрифт. Поэтому не стоит использовать бесплатные уникальные шрифты от дизайнеров студентов. Скорее всего они не слышали про то, что могут быть разные символы.

Культура написания кода

Культура написания кода

Культурного человека от некультурного отличает то, как легко другим с ним взаимодействовать, общаться, находится рядом. То же самое и с культурой написания кода. На фоне множества стандартов и требований к коду существует множество необязательных действий и манипуляций такие, как форма записи каскадных стилей, или расставления комментариев.

Комментарии

Если вы создаете личный блог, то, чисто теоретически, можете наплевать на «правило» писать комментарии к коду. Чисто теоретически – потому что разобраться самостоятельно гораздо легче спустя какое-то время в том, что же ты себе там написал.

Что же до коллективных проектов, то это просто обязательно. И кроме написания комментариев в коде, существует такое понятие, как документация. На моем опыте работа с площадкой ThemeForest, где ты делаешь продукт для международного общества это просто обязательство, которое тщательно проверяется ревизорами, отлично мотивирует. Мотивирует делать правильно, с меньшим числом правок и в принципе дисциплинирует.

Комментарии для верстки формально имеют один способ написания, для HTML это `<!-- Комментарий -->`, а для CSS это либо `//` в конце строки, либо `/* Комментарий */` в любом месте кода. Но! Вы можете отлично разнообразить вид комментария.

В случае работы с HTML полезно показывать начало и конец описываемой секции. Я это делаю следующим образом:

```
<!-- NAV section -->
<nav>
<ul>
<li><a href="#">Ссылка</a></li>
<li><a href="#">Ссылка</a></li>
</ul>
</nav>
<!-- NAV section END -->
```

С помощью флага END я указываю на конец секции.

Разница в деве и релизе

Если вы верстаете код, который будет передан, например, программисту, то определенно стоит расставить комментарии для себя, него. Если же вы являетесь конечным человеком, имеющим дело с версткой, то для релиза комментарии могут быть удалены. Так как в первую очередь они нужны именно для вас и вашей команды.

К слову, препроцессоры и шаблонизаторы позволяют оставлять в деве комментарии, которые не появятся в релизе автоматически. А в некоторых языках, по умолчанию, комментарии не уходят в релиз.

Например SASS:

```
// Этот комментарий не уйдет в релиз
```

```
/* А этот появится в релизе
```

Или язык PUG:

```
// Этот комментарий уйдет в релиз
```

```
//— А этот не уйдет туда
```

Конкретно в данном случае последние актуально использовать, если вы создаете файл типа data, где собираете демо-данные для вывода в шаблон. Так как в языке pug вы физически импортируете этот файл в макет, то все комментарии, что вы там оставите уйдут в конечный документ. Чтобы этого избежать такие комментарии стоит «экранировать».

Переносы строк, отступы

Среди кодеров (тут я беру в качестве понимания термина всех, кто пишет код) нередки случаи хаотичного коддинга, самый ужасный пример из которого являются переносы. Я имею в виду либо слишком много переносов, либо их отсутствие.

Я уверен, что код должен быть читаем: согласитесь, если в обычном тексте начнутся хаотические переносы – это создаст проблемы для восприятия. То же самое с отсутствием переносов, представьте, что вся эта книга будет написана одним абзацем. Звучит ужасно, не так ли?

Поэтому если вы хотите выделить код относительно другого – воспользуйтесь комментариями.

В качестве заключения

К счастью или сожалению, данные теоретические знания не смогут вам помочь стать супер-верстальщиком. В этом может помочь только титанический труд и опыт.

Данный же материал предназначен для того, чтобы брать иногда его в руки и проверять, не сбился ли ты с пути. Не стал ли создавать откровенный шлак, состоящий ради шлака внутри шлако-проектов.

На самом деле, действительно, очень легко пойти по пути наименьшего сопротивления. И я нередко срывался в эту легкость. Но каждый раз возвращался и старательно себя настраивал, чтобы уж точно так больше не делать.

Чего рекомендую и вам. Не создавайте плохие сайты и приложения. Делайте мир лучше, и такие проекты, на которых не стыдно через 5-10 лет иметь ссылку на свой сайт.

Я верстальщик

Веб-верстальщик

Арсений Матыцин

12+

