2.6. Частые проблемы из-за неправильной CSS-архитектуры проекта

~ 5 минут

Мы уже знаем, что из-за неправильной CSS-архитектуры проекта всё может сломаться, сейчас докажем.

Ситуация первая

На странице есть несколько похожих элементов, но в одном (или больше) случае элемент выглядит немного иначе.

Как обычно поступают: ищут родительский элемент (либо создают такой элемент искусственно), и для исключительного случая описывают следующие правила.

```
/* повторяющийся элемент */
.button {
  background-color: yellow;
  border: 1px solid black;
  color: black;
  width: 50%;
}

/* первый исключительный случай */
#sidebar .button {
  width: 200px;
}

/* второй исключительный случай */
body.homepage .button {
  background-color: white;
}
```

Какие проблемы могут появиться? В разметке элемент с классом `button` везде одинаковый, но при этом он почему-то выглядит по-разному в сайдбаре и на главной странице. То есть элемент ведёт себя непредсказуемо, другие разработчики не понимают, почему так происходит. Такой элемент плохо масштабируется: если такой виджет нужен в другом месте, то придётся добавлять ещё один селектор, чтобы реализовать это. А если дизайн или формат виджета изменится, стили нужно будет поменять в нескольких местах. В самом пессимистичном варианте эти места будут раскиданы по стилевому файлу.

К тому же такой код противоречит одному из принципов *SOLID* — открытости/закрытости, который говорит о том, что части *ПО* (*программного обеспечения*) должны быть открыты для расширения, но закрыты для изменения. Зачем родительскому блоку знать, какие блоки в него вложены? Если мы спроектируем систему так, что в родительский блок можно поместить что угодно, это будет более универсально и избавит нас от лишних связей.

Ситуация вторая

Усложняем селекторы, чтобы поднять специфичность правила. Такие селекторы, как правило, очень сильно зависят от HTML, что не очень хорошо, так как разметка в любой момент может измениться, а значит, придётся менять селекторы в стилях.

```
#main-nav ul li ul li div { }
#content article h1:first-child { }
#sidebar > div > h3 + p { }
```

Ситуация третья

Названия классов для элементов слишком общие.

На больших проектах высока вероятность, что простое и распространённое имя класса, как, например, `title`, будет использовано в другом контексте или даже само по себе. В примере ниже для заголовка карточки товара используется класс `title`. Но в проекте вполне может быть заголовок раздела с классом `title`. Много разных элементов с одинаковыми классами — верный путь к тому, чтобы забыть о каком-то из них при изменениях и получить настоящий баг.

```
<h2 class="title">Категория товаров</h2>
...
<article class="card">
  <h3 class="title">…</h3>
  <div class="contents">
    Содержимое…
    <button class="action">Жми сюда!</button>
```

```
  </div>
</article>
```

```
.title {
  color: #353a5a;
}

...

.card {...}
.card .title {
  font-size: 14px;
}
.card .contents {...}
.card .action {...}
```

Ситуация четвёртая

В одном правиле слишком много что определяется: и шрифт, и фон, и позиционирование, и другие параметры. Внешний вид можно использовать повторно, а раскладку и позиционирование уже нет, и если на странице (проекте) появится элемент с похожим оформлением, то придётся копировать стили. А это сложно поддерживать: чтобы изменить внешний вид, нужно вносить правки в множество правил вместо того, чтобы поправить одно-единственное правило, ответственное за внешний вид. Лучше использовать больше правил, которые отвечают за что-то одно (разделение ответственности).

```
.card {
  position: absolute;
  top: 20px;
  left: 20px;
  background-color: red;
  font-size: 1.5em;
  text-transform: uppercase;
}
```

CSS определяет, как выглядит ваш компонент, а HTML применяет этот вид к элементам на странице. Чем меньше CSS «знает» про структуру HTML, тем лучше.

Если проект планируется «одноразовым», например, для мероприятия, которое пройдёт и больше не повторится, то не так важно, как вы сделаете его сайт. Но если планируется развитие, расширение и поддержка, то лучше подстелить соломки, и сделать проект так, чтобы не получить кучу проблем из-за плохой архитектуры.

Прочитали главу?

Нажмите кнопку «Готово», чтобы сохранить прогресс.

Готово

Если вы обнаружили ошибку или неработающую ссылку, выделите ее и нажмите Ctrl + Enter

Поиск по материалам

Git

Все материалы

В самом начале

- [ ] Пройдите опрос
- [ ] Укажите персональные данные
- [ ] Изучите регламент
- [ ] Прочитайте FAQ
- [ ] Добавьте свой Гитхаб
- [ ] Выберите наставника
- [ ] Создайте проект

Мой наставник

Выбрать наставника

Работа с наставником

У вас осталось **10** из 10 консультаций.

История




Практикум

Тренажёры

Подписка

Для команд и компаний

Учебник по PHP

Профессии

Фронтенд-разработчик

JavaScript-разработчик

Фулстек-разработчик

Курсы

HTML и CSS. Профессиональная вёрстка сайтов

HTML и CSS. Адаптивная вёрстка и автоматизация

JavaScript. Профессиональная разработка веб-интерфейсов

JavaScript. Архитектура клиентских приложений

React. Разработка сложных клиентских приложений

Node.js. Профессиональная разработка REST API

Node.js и Nest.js. Микросервисная архитектура

TypeScript. Теория типов

Алгоритмы и структуры данных

Паттерны проектирования

Webpack

Vue.js 3. Разработка клиентских приложений

Git и GitHub

Анимация для фронтендеров

Блог

С чего начать

Шпаргалки для разработчиков

Отчеты о курсах

Услуги

Работа наставником

Для учителей

Стать автором

Информация

Об Академии

О центре карьеры

Остальное

Написать нам

Мероприятия

Форум

Участник