3. Подходы стилизации React приложений

Содержание урока

- Обзор;
- История;
- CSS;
- Инлайновая стилизация;
- Препроцессоры;
- CSS-модули;
- Постпроцессор PostCSS;
- Итог.

Обзор

Привет! 👋🏻 В этом уроке мы рассмотрим различные подходы стилизации React-приложений, а также разберём хорошие и плохие стороны каждого подхода. Благо, современный фронтенд предоставляет нам широкую свободу выбора, так что давай разбираться что подойдёт лучше, а что нет. 🦉

История

Давным давно, в древности, в тёмные 👹 века веб-разработки (в 90-х годах), веб-страницы были горами HTML, приправленного инлайн-стилями в одном файле. 🐵

```
<body style="background-color:powderblue;">
    <h1 style="color:blue;">Magic</h1>
    <p style="color:red;">Expecto Patronum!</p>
</body>
```

Данный подход считался мейнстримом до тех пор, пока олдскульные разработчики не столкнулись с проблемой поддержки эклемпляров файлов свыше, где разметка и стили определялись в виде единой ~~гигантской~~ контрукции. Поддерживать такое стало сложно. 🙀

Так, в 1996 году появился CSS с революционной миссией глобального переосмысления и наведения порядка. 👸🏼

Этот период тесно переплетён с появлением языка JavaScript в 1995 году. HTML получил двух ~~новых~~ друзей, и преобразился в новом свете. Очень скоро возник новый стандарт разделения ответственности между разметкой HTML, стилизацией CSS и логикой JavaScript.

Данное разделение устоялось в сознаниях разработчиков как стандарт де-факто. Но с появлением React, пришла новая эра фронтенда и определённые тенденции стали обращаться к истокам. 🧙

Хорошо это или плохо? Что сулят настроения современности?

Это нам и предстоит узнать. К делу! 👊🏽

CSS

Тут можно не стесняться 🦣 в комнате.

CSS в целом это хорошо только в случае его глубокой конфигурации с дополнительными инструментами, и никак иначе.

Дело в том, что в современном фронтенде приложения вырастают до невиданных ранее масштабов. Учитывая этот факт, мы можем выделить несколько критических минусов и недостатков CSS, мешающих рассматривать чистый CSS как достойного кандидата к использованию:

- В CSS, любой селектор глобален по умолчанию; 🤦
- Вложенные селекторы (`.main .sidebar .button` и т.д.) очень непрозрачны и сложны в понимании при больших масштабах; 🙈
- Проблема `Dead Code` — как узнать, можем ли мы удалить определённый блок/строку/инструкцию CSS и быть уверенными, что всё остальное будет везде хорошо работать? 🤵

Нативный CSS хорош в своей идее но ужасен на практике. Поддерживать веб-приложения, стилизированные с помощью чистого CSS — сущий ад. 🔥 Для осуществления, казалось бы, тривиальной задачи, порой приходится выделять неоправданно много времени и нервов.

Глобальная природа CSS — ночной кошмар даже самого смелого и опытного разработчика. Проблема вложенных селекторов делает ситуацию еще хуже. Если несколько блоков CSS имеют одинаковую «специфичность» (или «вес»), то выигрывает «последний», если только в первом (и еще в парочке мест, ~~ну очень надо было!~~) кое-кто не поленился прописать несколько инструкций `!important` (~~и никому не сказал об этом~~). В целом, приложения, написанные на обычном CSS становятся неподдерживаемыми очень быстро.

Естественно без «танцев ~~у костра с бутылкой рома~~ с дарбукой» тут не обойтись. Решений этих и не только проблем было придумано много, но большинство из них всего-лишь отдаляют неизбежное. 🤖

Приход React во фронтенд-разработку наделал немало шуму и спровоцировал существенные волнения даже в разрезе стилизации: стал набирать популярность подход стилизации приложения прямиком из JavaScript, прямо как в старые ~~добрые~~ времена.

Давай разузнаем что побуждает разработчиков следовать данному подходу, и чем он хорош, а чем — не очень. 🧐

Инлайновая стилизация

В самом простом виде подход инлайновой стилизации в React может выглядеть следующим образом.

🖥️ Пример кода 3.1:

```
class LectrumBuddy extends Component {
    render () {

        return (
            <h1
                style = { {
                    fontSize:    '1.5em',
                    paddingLeft: 5,
                    color:       'fireBrick'
                } }>
                I love Lectrum!
            </h1>
        );
    }
}
```

В прошлом уроке мы узнали о JSX и выражениях, которые можно объявить в фигурных скобках. Именно это и происходит в `примере кода 3.1` в `строке 6` — здесь объявлен литерал JavaScript-объекта, и привязан к атрибуту `style` элеманта `<h1>` в виде значения. Для описания свойств стилей элеманта `<h1>` используеться формат camelCase, то есть `font-size` становиться `fontSize`, а `padding-left` становиться `paddingLeft`.

Следуя инлайновому подходу стилизации хорошей практикой является вынесение описаний стилей из разметки для улучения читаемости.

🖥️ Пример кода 3.2:

```
1  export default class LectrumBuddy extends Component {
2      render () {
3          const style = {
4              fontSize:    '1.5em',
5              paddingLeft: 5,
6              color:       'fireBrick'
7          };
8
9          return <h1 style = { style }>I love Lectrum!</h1>;
10     }
11 }
```

В `примере кода 3.2` описание стилей вынесено в отдельную декларацию в `строке 3`.

🖐🏽 Совет бывалых:

> В 97.9% случаев возвращаемый компонентом JSX должен возвращать только `результат` вычислений, распечатывать только готовое значение. Чем меньше инструкций и всевозможных вычислений содержит JSX-разметка тем чище и понятней она есть. Максимальное вынесение логики и объявлений из разметки является хорошей практикой и способствует сохранению качества кода, времени и здорового сознания разработчика.

У данного подхода стилизации есть интересная особенность. JSX расценивает числа в объекте стилей как пиксели по-умолчанию. В `примере кода 3.2`, в `строке 5` React автоматически добавит суффикс `px` к значению `5` свойства `paddingLeft`. 👍🏽

Если необходима иная единица измерения, нежели `px` , например `проценты`, `rem`-ы, `em`-ы, или `vv`, суффикс нужно указывать явно, а всё значение должно стать строкой. 👩‍🎓

В `примере кода 3.2` мы провели небольшой рефакторинг, вынеся объект с описаниями стилей из разметки в привязку к отдельному идентификатору. В реальных проектах чаще всего приходится идти дальше и абстрагировать стили еще больше.

У нашего друга Оскара есть 🐈, его зовут Флаффи. Флаффи любит, когда его гладят. 👋🏽 Давай создадим компонент-кнопку, при нажатии на которую Флаффи будет получать желаемую ласку (кто-то его погладит, обязательно).

🖥 Пример кода 3.3:

```
// PetFluffy.js

import React, { Component } from 'react';
import Styles from './PetFluffy.css.js';

export default class PetFluffy extends Component {
    render () {

        return <button style = { Styles.button }>Pet Fluffy.</button>;
    }
}
```

```
// PetFluffy.css.js

export default {
    button: {
        backgroundColor: 'gold',
        width:           100,
        height:          25,
        borderRadius:    5,
        fontSize:        15
    }
};
```

Отделяя стили компонента в отдельный JavaScript-файл, экспортирующий объект с описанием этих стилей, мы во-первых улучшаем чисоту кода, а во-вторых уменьшаем когнитивную нагрузку при чтении логики разметки и стилей. А в-третьих используем потенциал `компонентной` и `переиспользуемой` природы React.

Мы можем переиспользовать компонент `PetFluffy` из `примера кода 3.3` в любой части приложения сколько угодно раз.

🖥️ Пример кода 3.4:

```
// CatSession.js

import React, { Component } from 'react';
import PetFluffy from './PetFluffy';

export default class CatSession extends Component {
    render () {

        return (
            <section>
                <div>Pet the cat. Meow.</div>
                <PetFluffy />
            </section>
```

```
14          );
15      }
16  }
```

Идея написания стилей в форме литералов объектов JavaScript приближает нас к самому языку JavaScript, используя весь потенциал этого языка программирования. Любую операцию, реализируемую с помощью JavaScript можно применить к инлайн-стилизации. Это очень удобно.

Плюс содержание стилей компонента в отдельном JavaScript-файле описывает определённый уровень инкапсуляции, что есть хорошо. 😇

Все эти бонусы в целом выглядят неплохо, но есть и тёмная сторона луны. 🌒

Следуя подходу инлайновой стилизации, нам недоступны (возможно пока) такие фичи CSS, как:

- псевдо-классы: `:hover`, `:active`;
- псевдо-элементы: `::first-line`, `::first-letter`, `::before`, `::afte`;
- медиа-запросы;
- различные директивы на подобии `@keyframes`;
- и еще многое другое.

Что-уж тут, это проблема. Ведь в современном мире все ~~никто~~ сильно сфокусированы на обеспечении отзывчивого и адаптивного дизайна, и без этих прелестных возможностей CSS не обойтись никак.

Плюс, когда дело доходит до сборки, деплоя и использования приложения, содержание стилей в ~~гигантском~~ JavaScript-файле приводит к неэффективному рендерингу страницы.

Содержание стилей в отдельном же файле, наоборот, приносит множество бонусов, так как файлы с CSS могут быть загружены параллельно с файлами JavaScript, и обработаны браузером с учётом множества оптимизаций.

Браузеры очень-очень стараются загрузить разметку, код и стили в наиболее эффективном виде. Содержание стилей в отдельном файле ускорит изначальную загрузку страницы.

Естественно, на все перечисленные выше проблемы давно существует множество решений.

Инструменты, стоящие рассмотрения:

- 💅 styled-components;
- 👩‍🎤 emotion;
- 🐓 aphrodite;
- ☢️ radium.

Но всё это возвращает нас к проблеме CSS — мы всё ещё боремся с `последствиями`. Поэтому пока-что мы можем расценивать инлайновый подход стилизации как молодой и перспективный, но всё еще не несущей в себе достаточно аргументов для его использования в полной мере.

Препроцессоры

Препроцессоры — попытка реализации CSS как полноценного языка программирования. Препроцессорый CSS являет собой по сути компилируемое CSS-расширение.

Мы рассмотрим препроцессор sass как пример, хотя существуют и другие хорошие альтернативы:

- stylus;
- less.

Препроцессор — это коробочное решение, дающее широкий спектр функциональных возможностей в стилизации, таких как:

- переменные;
- вложенность;
- миксины;
- математика;
- функции;
- переборы;
- интерполяция и конкатенация;
- наследование;
- модульная система импортов.

Препросессоры уникальны тем, что являются намного более мощным инструментом, чем обычный CSS, предосталяя возможность использовать силу языка программирования, при этом, по сути, в основе оставаясь CSS.

Например, препроцессор `sass` позволяет нам использовать следующие инструкции.

🖥️ Пример кода 3.5:

```
@import 'reset-css';

$primary-font:  roboto, sans-serif;
$primary-color: #333;

body {
    color: $primary-color;
    font:  100% $font-stack;
}
```

```
10
11  nav {
12      ul {
13          margin:     0;
14          padding:    0;
15          list-style: none;
16      }
17
18      li {
19          display: inline-block;
20      }
21
22      a {
23          display:         block;
24          padding:         6px 12px;
25          text-decoration: none;
26      }
27  }
```

В `примере кода 3.5` мы использовали следующие возможности `sass`:

- В `стоке кода 1` — невероятно мощную директиву `@import`, подключающую любой внешний файл со стилями: `.scss` или `.css`;
- В `строках кода 3 и 4` — переменные, открывающие нам возможность объявить значение раз и переиспользовать это значение сколько угодно раз. Это решает проблему, когда дизайнер решил увеличить размер шрифта на 1 пиксель и нужно править это изменение в 100 файлах. С переменными это можно сделать только один раз;
- В `строке кода 12` — вложенность в явном виде. Такая вложенность повышает читаемость кода и понимание происходящего в разы.

В целом препроцессорное решение долгое время решало значительное колличество проблем и являлось предпочтительным. Но у препроцессоров, несмотря на всю их мощь, всё же есть свои недостатки.

Самая главная — препроцессоры не решают проблему глобальной природы CSS. Плюс препроцессор — дополнительный шаг при сборке и в некоторых случаях это может быть медленно.

К счастью некоторое время назад появилось весьма интересное решение глобальности CSS — `css-модули`. Эту технологию можно совмещать с препроцессорами, решая недостаток описанный выше. 🕵🏻

CSS-модули

CSS-модуль — это файл с CSS, в котором все имена классов и анимаций инкапсулированы внутри этого файла. При импортировании CSS-модуля в файл с React-компонентом, мы получим объект с привязкой локальных имён классов к глобальным. Глабальные имена классов присваиваются элементам в приложении в момент сборки, и всегда гарантированно уникальны. 👌

Использование CSS-модуля может выглядеть следующим образом.

🖥 Пример кода 3.6:

```
// PetCatButton.js

import React, { Component } from 'react';
import Styles from './styles.css';

export default class PetCatButton extends Component {
    render () {

        return <button className = { Styles.button }>Pet the cat.
</button>;
    }
}
```

```
/* styles.css */

.button {
    width:            100px;
    height:           25px;
    border-radius:    5px;
    background-color: darkOrange;
    font-size:        14px;
}
```

В HTML результат будет следующим:

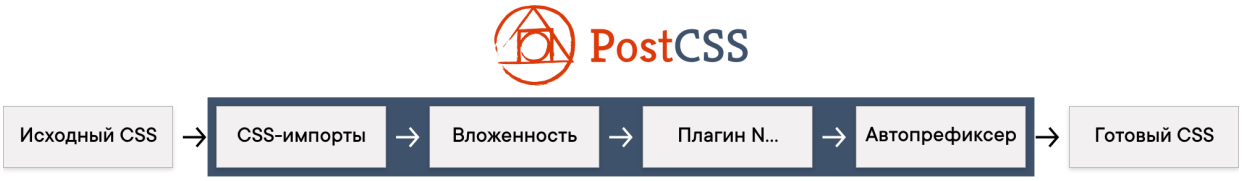
```
<button class = "Button__button__1DA66">Pet the cat.</button>
```

В `примере кода 3.6`, в `строке 4` в файле `PetCatButton.js` мы импортировали объект с привязкой локальных имён классов к глобальным. Это значит, что в `строке кода 9`, обратившись к свойству `button` идентификатора `Styles`, мы даём инструкцию привязать стили класса `.button` к элементу `<button>`. Имя класса преобразуется в связку имени класса и хеша, что обеспечаивает его уникальность: `Button__button__1DA66`.

Данный подход решает проблему глобальности CSS, инкапсулируя наборы стилей в модули. И хоть это весьма хорошее решение можно применять вместе с препроцессорами, существует более современная альтернатива, еще недавно обещавшая стать многообещающей, но сегодня это претендент на позицию общепринятого стандарта — `постпроцессор PostCSS`. 🎉

Постпроцессор PostCSS

PostCSS — интрумент для преобразования CSS с помощью JavaScript. Давай рассмотрим следующую схему.



`PostCSS` похож на обычный препроцессорный подход, за исключением того, что `PostCSS` — модульный. Обычный препроцессор предоставляет коробочное решение из которого разработчик, вероятно, будет использовать 10-30% доступных возможностей. Процесс сборки стилей с помощью `PostCSS` настраивается посредством установления цепочки `PostCSS-плагинов`, где каждый плагин — отдельная фича.

❗ Важно:

> При определении цепочки плагинов PostCSS порядок имеет значение! `CSS-импорты` → `Вложенность` и `Вложенность` → `CSS-импорты` потенциально могут дать разный результат. Будь осторожен при составлении последовательности цепочки плагоинов PostCSS.

Хочешь использовать `CSS-переменные`? Пожалуйста.

Хочешь супер-модные градиенты из будущего в виде `PostCSS-плагина`? Такое тоже есть.

Для каждой супер-интересной фичи есть свой плагин: вложенность, импорты, color-функции.

Существует также невероятно мощный плагин Autoprefixer, автоматически выставляющий вендорные префиксы для всего CSS в зависимости от браузеров, которые необходимо поддерживать. Всё что нужно сделать — описать конфигурационный файл `.browserslistrc`, и `Autoprefixer` сделает всю грязную работу:

```
Chrome 58
Safari 10.2
Firefox 40
```

✋ Совет бывалых:

> Существует также особый супер-набор самых популярных и полезных плагинов, соответствующих W3C-спецификации, называемый cssnext.

`PostCSS` очень популярен, экосистема плагинов постоянно растёт и улучшается, поэтому на данный момент, подход стилизации с помощью `постпроцессора PostCSS` можно считать самым современным и многообещающим. 🥇

Итог

Мы разобрали самые популярные на сегодняшний день подходы стилизации React приложений. Какой из них выбрать? Это нужно решить тебе самому, учитывая плюсы и минусы каждого, а также свои потребности. А пока давай повторим преимущества и недостатки каждого подхода. 🙌🏽

CSS

👍 Преимущества:

- Лёгок в использовании;
- Работает сразу, не нужно настраивать.

👎 Недостатки:

- Глабален по-умолчанию;
- Отсутствие удобной вложенности;
- Очень быстро становится неподдерживаемым.

Инлайновый подход стилизации

👍 Преимущества:

- Предоставляет всю силу языка JavaScript.

👎 Недостатки:

- Отсутствие поддержки псевдо-классов, псевдо-элементов, медиа-запросов, css-директив и многого другого;
- Нарушение стандарта W3C;
- Предвещает потенциальные проблемы при потреблении конечным пользователем ведь обычный CSS обрабатываеться браузером лучше.

Препроцессоры

👍 Преимущества:

- Предоставляют мощь языка программирования, не нарушая правил W3C;

👎🏽 Недостатки:

- Радикален и часто предоставляет намного больше, чем нужно;
- В некоторых случаях существенно замедляет процесс сборки;
- Подход достаточно устаревший на фоне технологий будущего.

PostCSS + CSS-модули

👍🏽 Преимущества:

- Отличный архитектурный дизайн, быстро работает;
- Модульный конструктор: бери только тот функционал, который необходим, и ничего более;
- Большой выбор плагинов, которые постоянно улучшаются и развиваются;
- Большое растущее комьюнити разработчиков;
- Легко написать свой плагин.

👎🏽 Недостатки:

- Для раскрытия всего потенциала необходимо использовать с CSS-модулями. На это тоже есть плагин;
- Настройка пайплайна плагинов — процесс весьма увлекательный и можно утонуть в нём на несколько суток. 🤷🏼‍♀️

Воу, ты уже прошел урок! В следующем мы коснёмся темы наполнения React-приложения данными с помощью механизма, называемого «пропсы» (`props`).

> Мы будем очень признательны, если ты оставишь свой фидбек в отношении этой части конспекта на нашу электропочту `hello@lectrum.io`.