Лекция 4

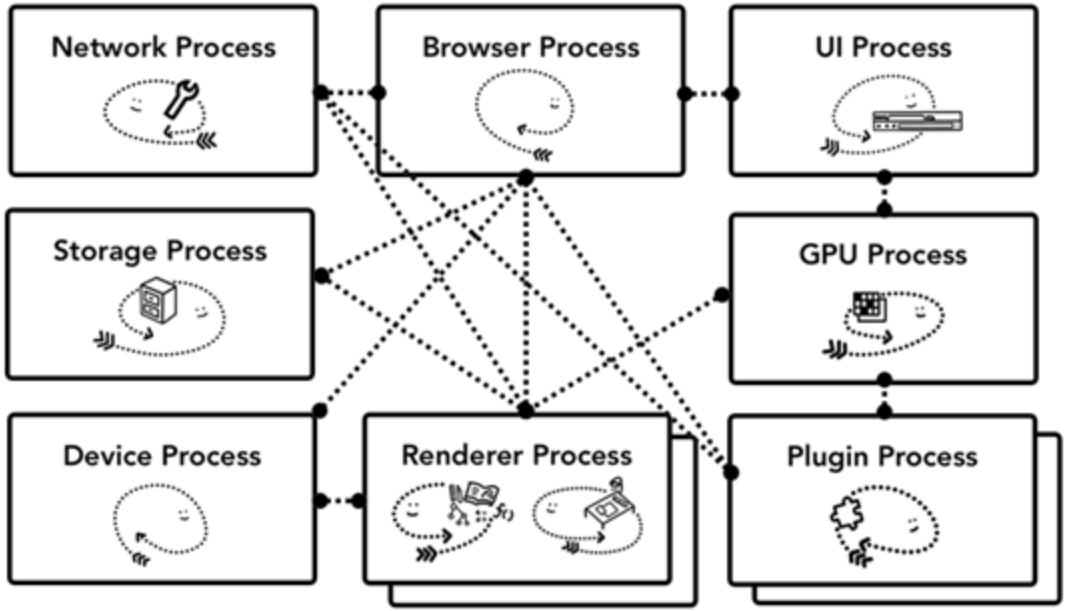
Архитектура Web

Что находится внутри браузера?

Сначала браузер загружает **только основную структуру HTML-страницы**.

Затем последовательно проверяет все теги и отправляет дополнительные GET-запросы для получения с сервера различных элементов — картинки, файлы, скрипты, таблицы стилей и так далее.
Поэтому по мере загрузки страницы браузер и сервер продолжают обмениваться между собой информацией.

Как только рендеринг завершен — пользователю отобразится полностью загруженная страница сайта.



Критические этапы рендера

Критические этапы рендеринга (Critical Rendering Path) - это последовательность шагов, которые выполняет браузер, когда преобразуется HTML, CSS и JavaScript в пиксели, которые вы видите на экране.

1. Загрузка HTML;
2. DOM;
3. CSSOM;
4. Дерево рендера (render tree);
5. Компоновка (layout);
6. Отрисовка (paint).

Подробнее тут https://developer.mozilla.org/ru/docs/Web/Performance/Critical_rendering_path

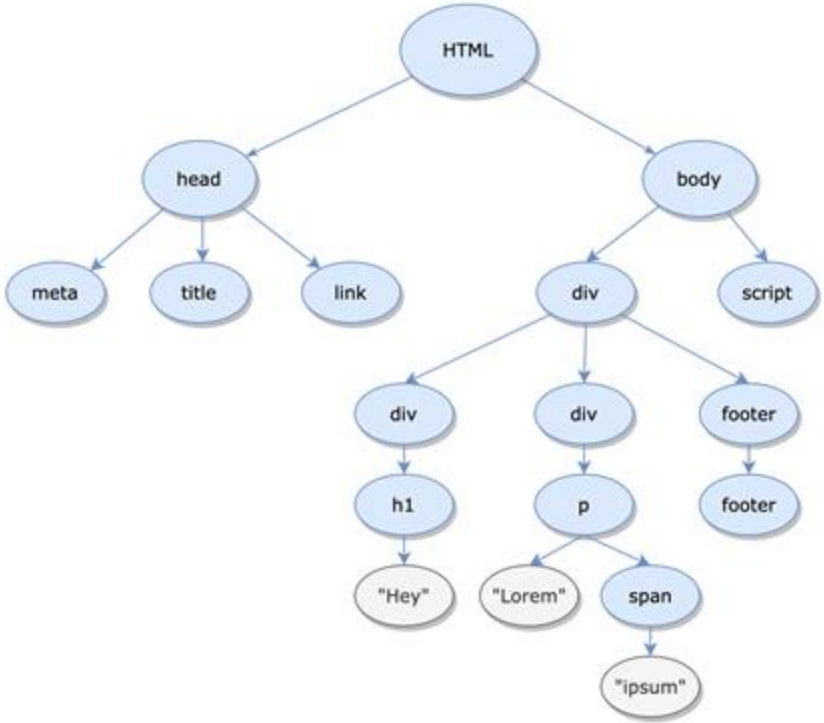
Document Object Model

Ответ в виде HTML превращается в токены, которые превращаются в узлы (nodes), которые формируют DOM дерево.

Простейший узел начинается с startTag-токена и заканчивается токеном endTag.

Узлы содержат всю необходимую информацию об HTML-элементе, соответствующем этому узлу.

Узлы (nodes) связаны с Render Tree с помощью иерархии токенов: если теги внутри других, то мы получаем узел (node) внутри узла (node), то есть получаем иерархию дерева DOM.



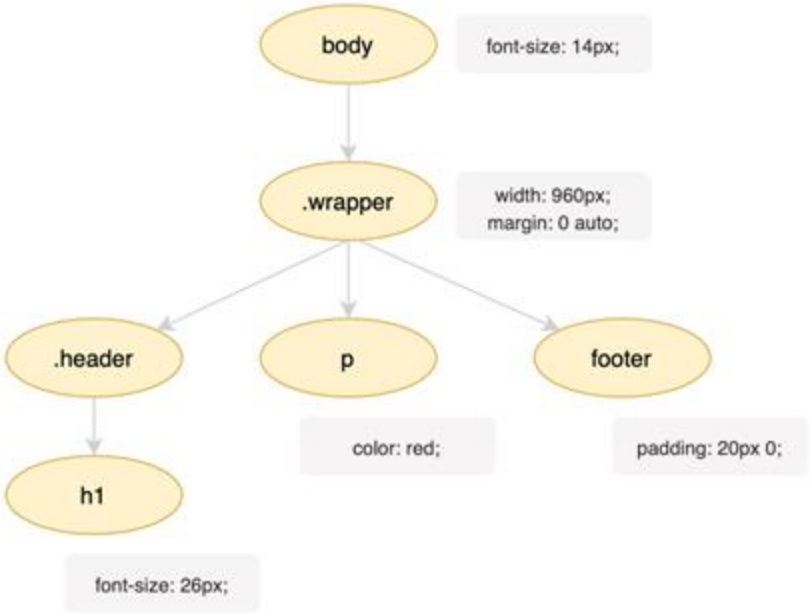
CSS Object Model

DOM несёт в себе всё содержимое страницы, а CSSOM содержит все стили страницы, то есть данные о том, как стилизовать DOM.

Если формирование DOM инкрементально, CSSOM - нет.

CSS блокирует рендер: браузер блокирует рендеринг страницы до тех пор, пока не получит и не обработает все CSS-правила.

Необходимо дождаться построения CSSOM, чтобы убедиться в отсутствии дополнительных переопределений.



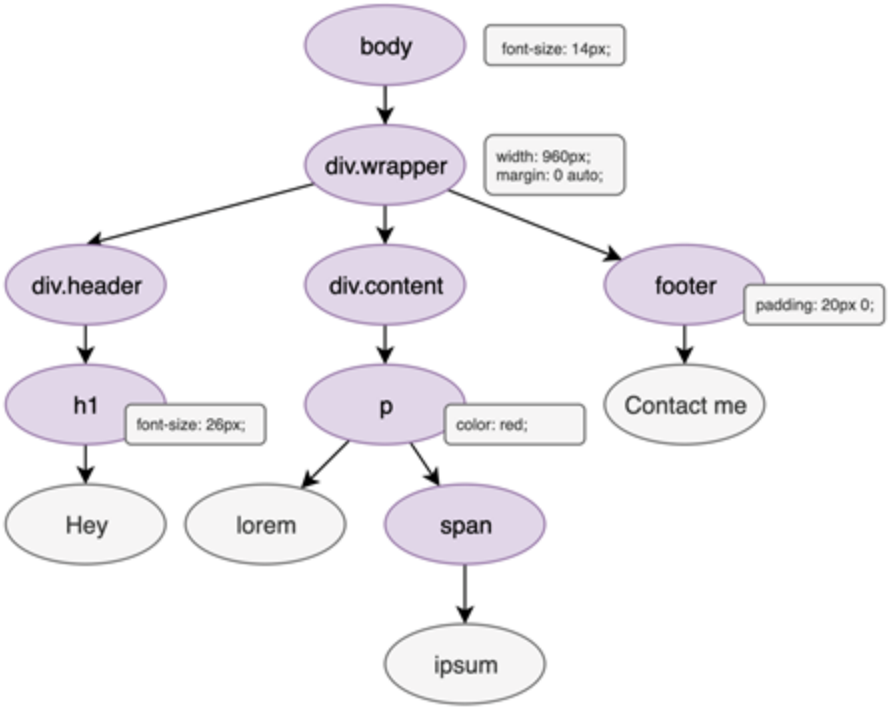
Дерево рендера (Render Tree)

Дерево рендера охватывает сразу и содержимое страницы, и стили: это место, где DOM и CSSOM деревья комбинируются в одно дерево.

Для построения дерева рендера браузер проверяет каждый узел (node) DOM, начиная от корневого (root) и определяет, какие CSS-правила нужно присоединить к этому узлу.

Дерево рендера охватывает только видимое содержимое, например, секция head (в основном) не содержит никакой видимой информации, а потому может не включаться в дерево.

Кроме того, если у какого-то узла стоит свойство display: none, оно так же не включается в дерево (как и потомки этого узла).



Компоновка (layout)

В тот момент, когда дерево рендера (render tree) построено, становится возможным этап компоновки (layout).

Layout — это рекурсивный процесс определения положения и размеров элементов из Render Tree.
Он начинается от корневого Render Object, которым является, и проходит рекурсивно вниз по части или всей иерархии дерева высчитывая геометрические размеры дочерних render object'ов.
Корневой элемент имеет позицию (0,0) и его размеры равны размерам видимой части окна.

К концу процесса layout каждый render object имеет свое положение и размеры.

осталось только нарисовать

Отрисовка (paint)

Последний этап в нашем списке - **отрисовка (paint)** пикселей на экране

Когда дерево рендера (render tree) создано, компоновка (layout) произошла, пиксели могут быть отрисованы.

При первичной загрузке документа (onload) весь экран будет отрисован.

После этого будут перерисовываться только необходимые к обновлению части экрана.

Ограничения JS в браузере

- Нельзя взаимодействовать с файловой системой
- Нет доступа к сетевым функциям, кроме того, что предоставляет сам браузер
- Нет возможности организовывать многопоточные вычисления. Есть воркеры, но они имеют определенные ограничения
- Нельзя создавать новые процессы / запускать программы (открытие новых вкладок не считается)

Npm

Node.js - исполнение JavaScript кода НЕ в браузере (взаимодействие с устройствами ввода-вывода, подключение внешних библиотек на разных языках, вызывая их из JavaScript-кода)

https://github.com/iu5git/Web/blob/main/tutorials/install/README.md

Подробнее тут https://nodejs.dev/

Npm - Node Package Manager

Менеджер пакетов, входящий в состав Node.js.

NPM состоит из двух основных частей:

- инструмент CLI (command-line interface – интерфейс командной строки) для публикации и загрузки пакетов
- онлайн-репозиторий, в котором размещаются пакеты JavaScript.

Как подключить библиотеки

- **Вариант с CDN**

```
<script src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.11.6/dist/umd/popper.min.js"></script>
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha2/dist/js/bootstrap.min.js"></script>
```

- **Вариант с Npm (**Скачиваем нужные нам зависимости)

```
npm install bootstrap@5.3.0-alpha2
```

- У нас появляется директория node_modules со всеми зависимостями
- Подключаем нужный нам пакет

```
import { ... } from 'bootstrap'
```

Npm (создаем свой модуль)

`npm init` - инициализируем свое приложение

- `package.json` - файл с описанием приложения
- `package-lock.json` - файл с версиями зависимостей

```
{
  "name": "awesome",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "bootstrap": "^5.2.3"
  },
  "devDependencies": {
    "typescript": "^4.9.5"
  }
}

```

Архитектурные решения

Как определить, является ли какое-то решение архитектурным. Как определить качество архитектурного решения?
Задайте себе вопрос:

А что если я ошибся и мне придется изменить это решение в будущем? Какие будут последствия?

Если некоторое решение размазано ровным слоем по всему приложению, то стоимость его изменения будет огромной, а значит это решение является архитектурным

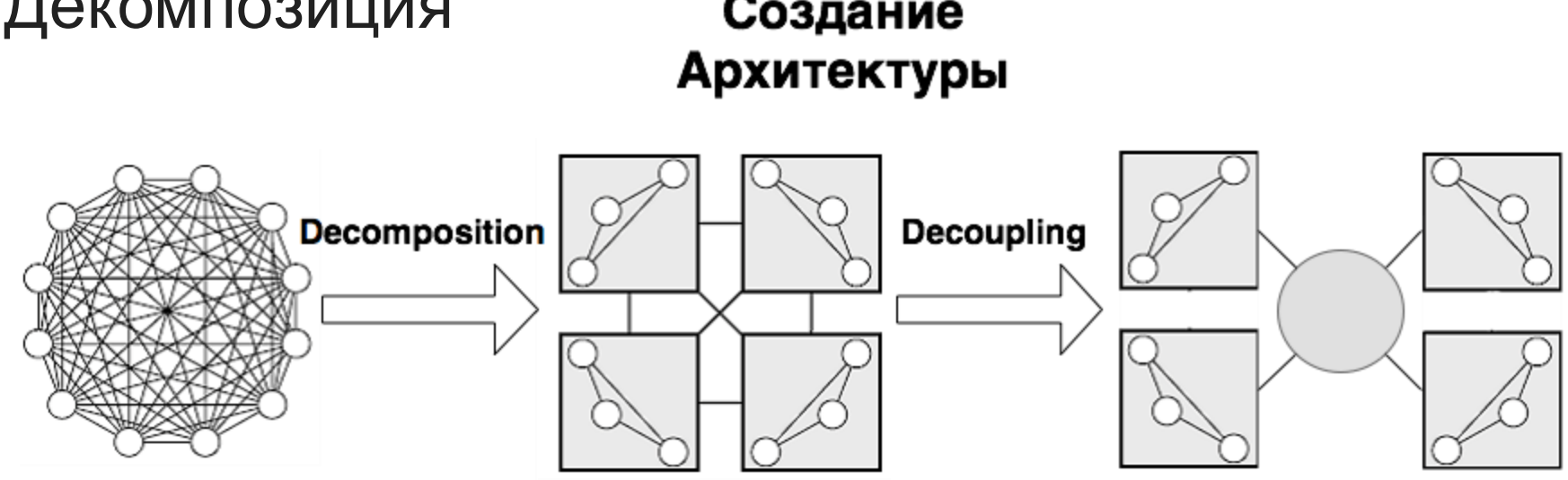
Методологии и принципы

- **DRY** — don’t repeat yourself (не повторяйте себя)
- **KISS** — keep it simple stupid (делайте вещи проще)
- **S.O.L.I.D.** — пять основных принципов объектно-ориентированного программирования и проектирования

Вы можете запоминать каждое слово отдельно
Вы можете дублировать каждый раз код

Но лучше запомнить простое правило произношения
И переиспользовать одинаковый код/разметку

Декомпозиция



- **Internal Cohesion** — сопряженность или «сплоченность» **внутри модуля** (составных частей модуля друг с другом)
- **External Coupling** — связанность **взаимодействующих друг с другом модулей**.

Декомпозиция (функциональная)

Во главе функциональной декомпозиции лежит паттерн Модуль

***Модуль** — это Функция + Данные, необходимые для её выполнения*

- Инкапсуляция данных
- Явное управление зависимостями (создание чёткой структуры подключаемых модулей)
- Уход от засорения глобального контекста

Декомпозиция (функциональная)

```
1 // user.js
2
3 const user = {}
4
5 export function setUser (newUser) {
6   user = newUser
7 }
8
9 export function getUser () {
10   return user
11 }
```

- **ОДИН файл — ОДИН модуль**
- Не допускать *циклические* зависимости

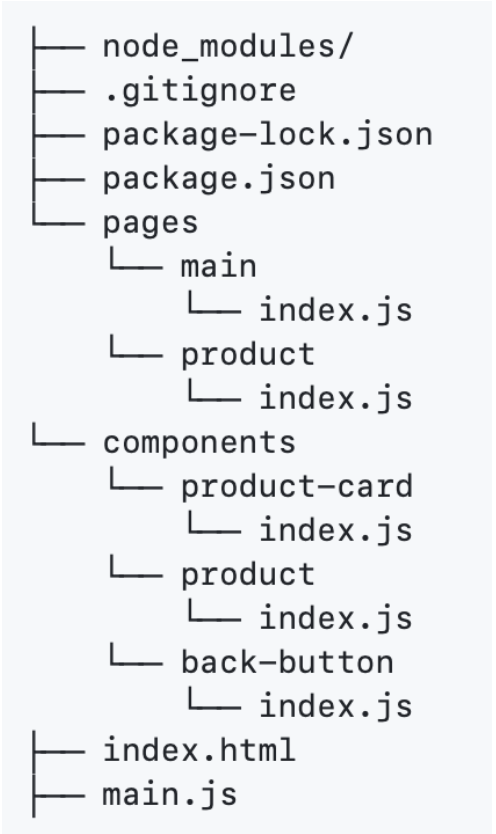
```
1 // func.js
2
3 import { getUser } from './user.js'
4
5 alert( getUser() );
6
```

Проектирование системы

- Какую функцию выполняет каждый модуль?
- Насколько модули легко тестировать?
- Возможно ли использовать модули самостоятельно или в другом окружении?
- Как сильно изменения в одном модуле отразятся на остальных?

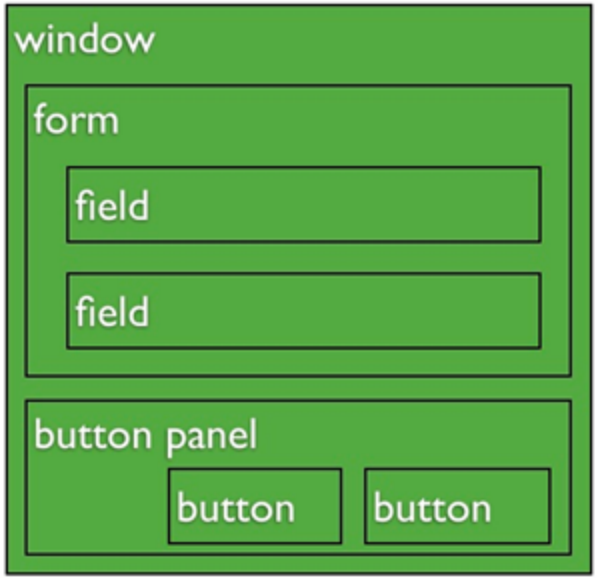
Критерии *хорошего* дизайна системы

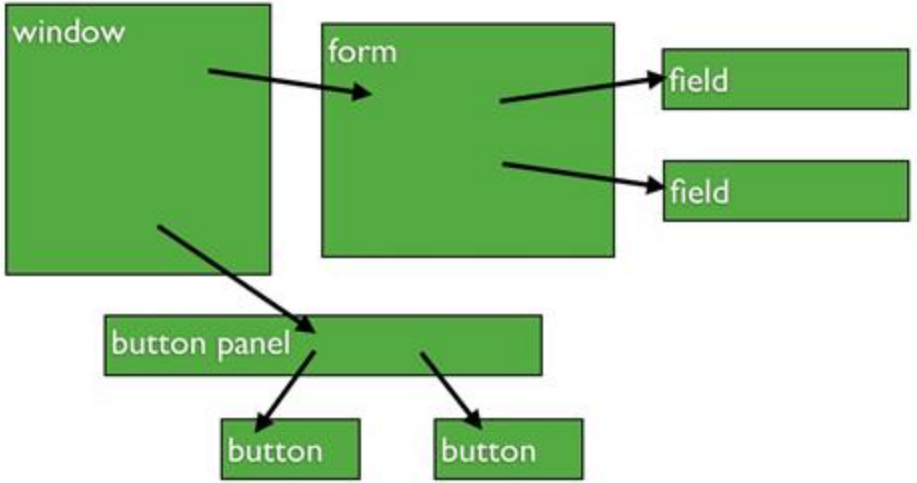
- Эффективность системы
- Гибкость и расширяемость системы
- Масштабируемость процесса разработки
- Тестируемость и сопровождаемость
- Возможность повторного использования



Декомпозиция (компоненты)

Компонентный подход - разделение кода приложения на независимые, слабосвязанные и переиспользуемые компоненты





Декомпозиция (советы)

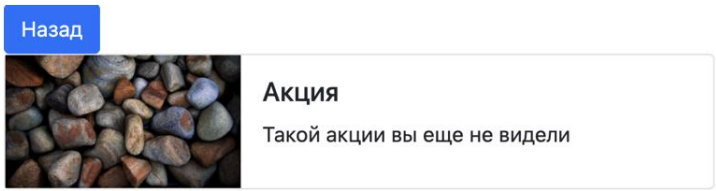
Компоненты могут быть **достаточно сложны внутри**, но они должны быть просты для использования **снаружи**

Компонентом может быть **вообще всё что угодно**, что выполняет какую-то функцию в вашем приложении

```
render() {
    this.parent.innerHTML = ''
    const html = this.getHTML()
    this.parent.insertAdjacentHTML('beforeend', html)

    const backButton = new BackButtonComponent(this.pageRoot)
    backButton.render(this.clickBack.bind(this))

    const data = this.getData()
    const stock = new ProductComponent(this.pageRoot)
    stock.render(data)
}
```



Шаблоны MVC

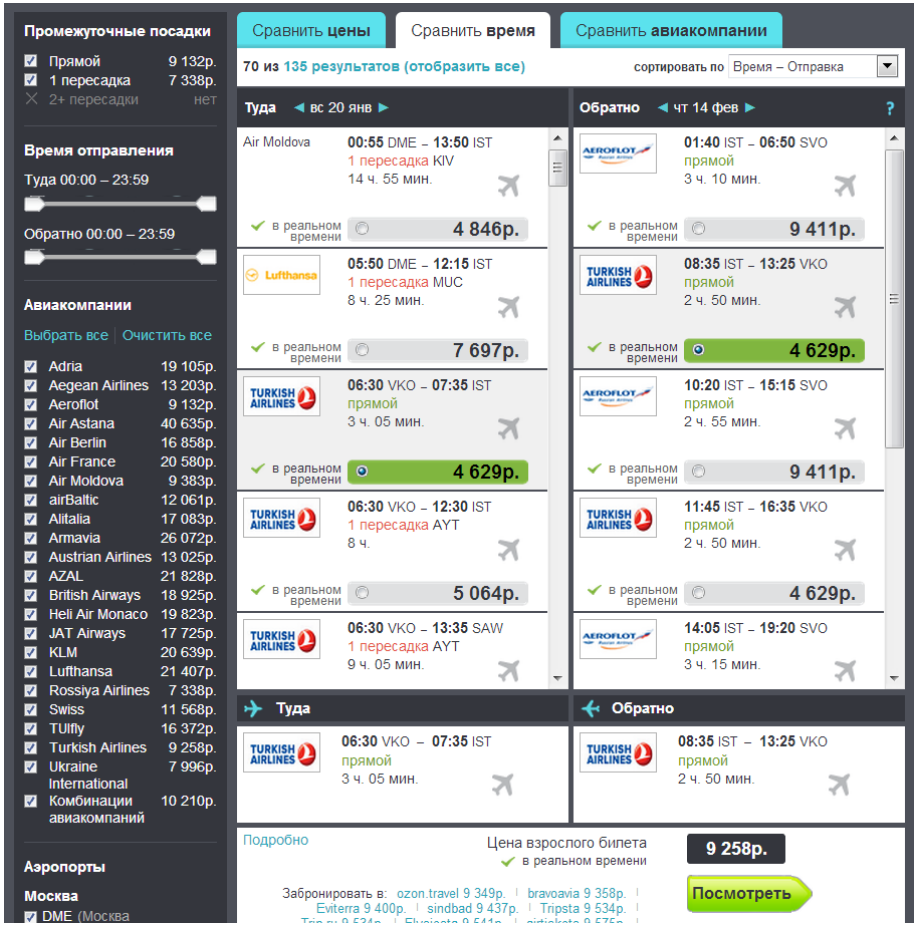
MV?

Или почему Китай не
отказывается от иероглифов?

Фильтрация и группировка

- У нас есть **список билетов**, но мы хотим их использовать по разному
- Показать самые **дешевые**, самые **быстрые**, другие пользовательские **фильтры**
- **Сгруппировать** по странам, по пересадкам, по дням
- **Сортировка**, дополнительные представления (**графики**, **гистограммы**)

- Выход: отделить **данные** и их **представление**



Япония, Токио

東京 日本

dōngjīng Rìběn

東京 日本

Tōkyō, Nihon

- Иероглифы означают понятие (Model)

- В разных регионах и странах пишут одно и тоже

- Но произносятся совсем по-разному (View)

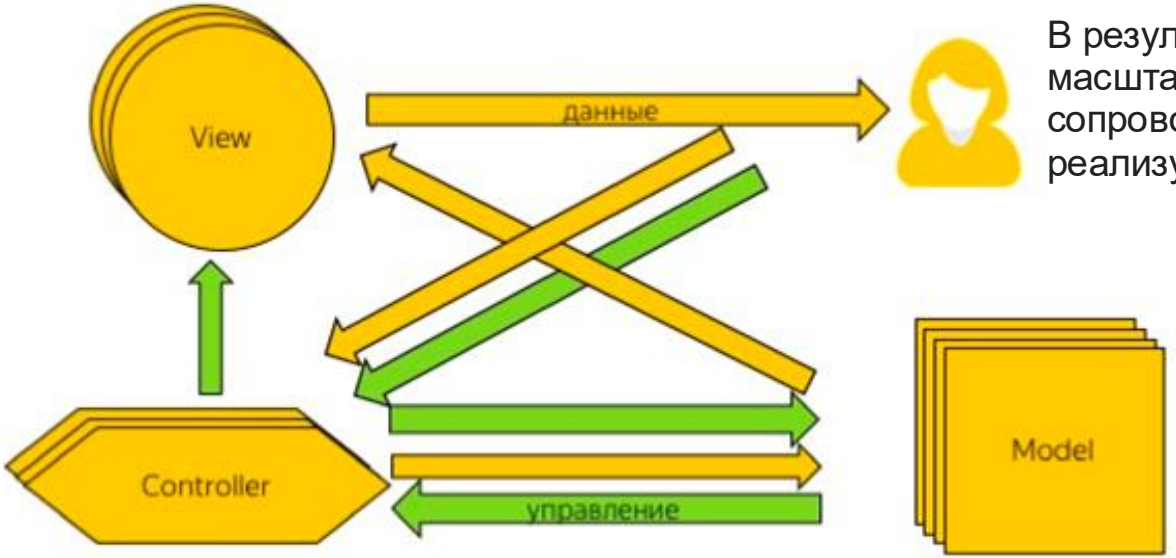
Шаблоны MVC

Шаблон MVC (Модель-Вид-Контроллер или Модель-Состояние-Поведение)

описывает простой способ построения структуры приложения, целью которого является отделение бизнес-логики от пользовательского интерфейса.

В результате, приложение легче масштабируется, тестируется, сопровождается и, конечно же, реализуется

Подробнее тут https://frontend.tech-mail.ru/slides/s5/



Модели MVC

- Содержит **бизнес-логику** приложения: методы для получения и обработки данных
- Не взаимодействуют с напрямую пользователем
- Не генерируют никакого HTML (**не управляют отображением данных**)
- Модели могут хранить в себе данные и они могут взаимодействовать с другими моделями

```
getData() {
    return [
        {
            id: 1,
            src: "https://i.pinimg.com/originals/c9/ea/65/c9ea654eb3a7398b1f702c758c1c4206.jpg",
            title: "Акция",
            text: "Такой акции вы еще не видели 1"
        },
        {
            id: 2,
            src: "https://i.pinimg.com/originals/c9/ea/65/c9ea654eb3a7398b1f702c758c1c4206.jpg",
            title: "Акция",
            text: "Такой акции вы еще не видели 2"
        },
```

Представления MVC

- Отвечают за **отображение данных**, содержат в себе вызовы шаблонизаторов, создание блоков и компонентов и всего такого
- Получают данные от напрямую от моделей или от контроллеров
- Взаимодействуют с моделями посредством контроллеров
- Являются посредниками между пользователем и контроллером

```
getHTML() {
    return (
        `
            <div class="card" style="width: 300px;">
                <img class="card-img-top" src="https://i.pinimg.com/original
                <div class="card-body">
                    <h5 class="card-title">Акция</h5>
                    <p class="card-text">Вот тут информация об акции</p>
                    <button class="btn btn-primary"">Нажми на меня</button>
                </div>
            </div>
        `
    )
}
```

```
render() {
    const data = this.getData()
    const productCard = new ProductCardComponent(this.parent)
    productCard.render(data)
}
```

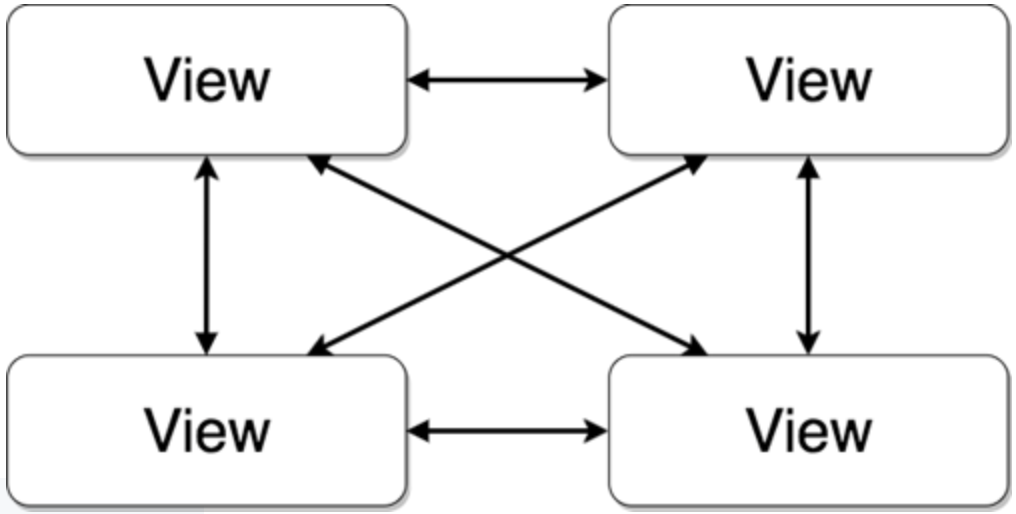
Контроллеры MVC

- **Являются связующим звеном приложения**
- Реализуют взаимодействие между вьюхами и моделями и взаимодействие вьюх друг с другом
- Должны содержать минимум бизнес-логики и быть максимально простыми в конфигурации (для возможности удобного изменения и расширения приложения)
- **Логика контроллера довольно типична и большая ее часть выносится в базовые классы**, в отличие от моделей, логика которых, как правило, довольно специфична для конкретного приложения

Собираем MVC

- Создаём базовый класс **View**
- Наследуем от него **MenuView**, **SignView**, **ScoreboardView**...
- Во вьюхах описываем описываем отображение определённой части приложения
- Содержимое **View** генерируется с помощью шаблонизатора
- Далее на основе уже имеющейся разметки создаются блоки и компоненты, либо они генерируются динамически на каком-то этапе жизненного цикла приложения
- Каждый момент времени активна только одна **View**

Роутинг

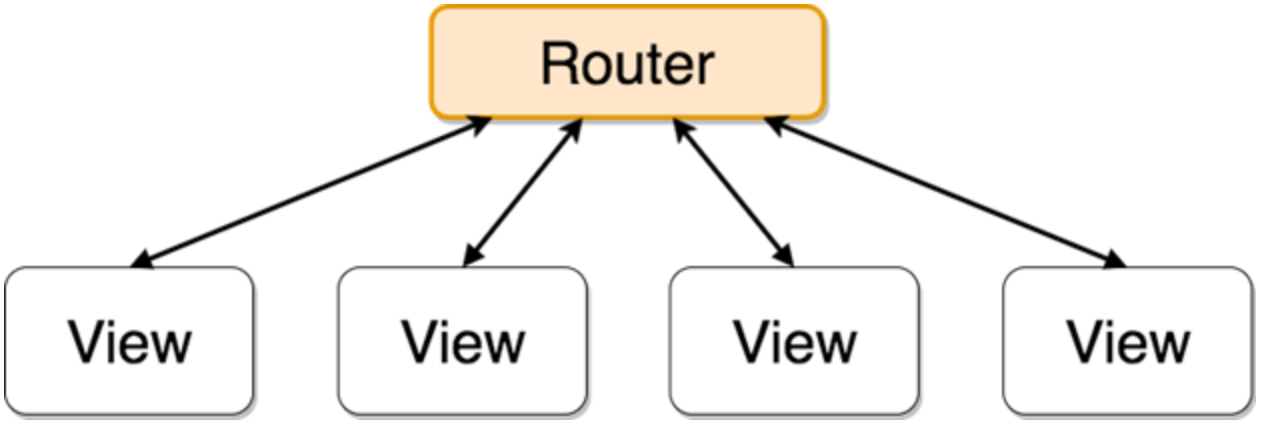



```
clickBack() {
    const mainPage = new MainPage(this.parent)
    mainPage.render()
}
```

```
clickCard(e) {
    const cardId = e.target.dataset.id

    const productPage = new ProductPage(this.parent, cardId)
    productPage.render()
}
```

Роутер (паттерн медиатор)



На сервере **роутинг** — это процесс определения маршрута внутри приложения в зависимости от запроса. Проще говоря, это поиск контроллера по запрошенному URL и выполнение соответствующих действий

На клиенте роутинг позволяет установить соответствие между состоянием приложения и **View**, которая будет отображаться. Таким образом, роутинг — это вариант реализации паттерна "Медиатор" для MV* архитектуры приложений

Что такое роутинг?

Таким образом взаимодействие и переключение между **View** происходит посредством роутера, а сами **View** друг о друге ничего не знают

```
1 Router.register('/', MenuView);
2 Router.register('/signup', SignupView);
3 Router.register('/scores/pages/{page}', ScoreboardView);
4 ...
5 Router.go('/scores/page/3'); // переход на 3 страницу (пагинация)
6
```

History API

Кроме этого, роутеры решают ещё одну очень важную задачу. Они позволяют эмулировать **историю переходов** в SPA-приложениях

History API — браузерное API, позволяет манипулировать историей браузера в пределах сессии

```
1 // Перемещение вперед и назад по истории
2 window.history.back(); // работает как кнопка "Назад"
3 window.history.forward(); // работает как кнопка "Вперёд"
4
5 window.history.go(-2); // перемещение на несколько записей
6 window.history.go( 2);
7
8 const length = window.history.length; // количество записей
9
```

Архитектура CSS

Подробнее тут https://frontend.tech-mail.ru/slides/s5/

Как стили попадают на страницу

- Браузерные стили
- link rel=”stylesheet”
- тег style
- атрибут style

```
/* Селекторы! */

*                                   /* универсальный селектор */
div, span, a                        /* селекторы по имени тегов */
.class                              /* селекторы по имени классов */
#id                                 /* селекторы по идентификаторам */
[type="text"], [src*="/img/"]       /* селекторы по атрибутам */
:first-child, :visited, :nth-of-type(An+B), :empty ...
::before, ::placeholder, ::selection, ::first-letter ...
a > a, a + a , a ~ a                /* вложенность и каскадирование */

```

Какие могут быть проблемы?

Задача: один и тот же компонент должен выглядеть по-разному в зависимости от страницы


```
1 /* Изменение компонентов в зависимости от родителя */
2 .button { border: 1px solid black; }
3 #sidebar .button { border-color: red; }
4 #header .button { border-color: green; }
5 #menu .button { border-color: blue; }
6
```

Какие могут быть проблемы?

Задача: найти элемент на странице "в слепую"

Проблема: сильная связанность со структурой документа

```
1 /* Глубокая степень вложенности */
2 #main-nav ul li ul li ol span div { ... }
3 #content .article h1:first-child [name=accent] { ... }
4 #sidebar > div > h3 + p a ~ strong { ... }
5
```

Какие могут быть проблемы?

Задача: сделать стили более понятными для всех

Проблема: пересечение имён с внешними библиотеками или даже внутри собственного проекта

```
1 /* Широко используемые имена классов */
2 .article { ... }
3 .article .header { ... }
4 .article .title { ... }
5 .article .content { ... }
6 .article .section { ... }
7
```

Признаки хорошей архитектуры

- **Предсказуемость** — изменение текущих стилей не ломает проект
- **Масштабируемость** — добавление новых стилей не ломает проект
- **Поддержка** — все в команде понимают, как писать стили
- **Повторное использование** — DRY

Объектно-ориентированный CSS

Разделение структуры и оформления: если есть общие стили оформления, то выносим их в отдельный класс

Зачем? - При изменении цветовой палитры правим в одном месте

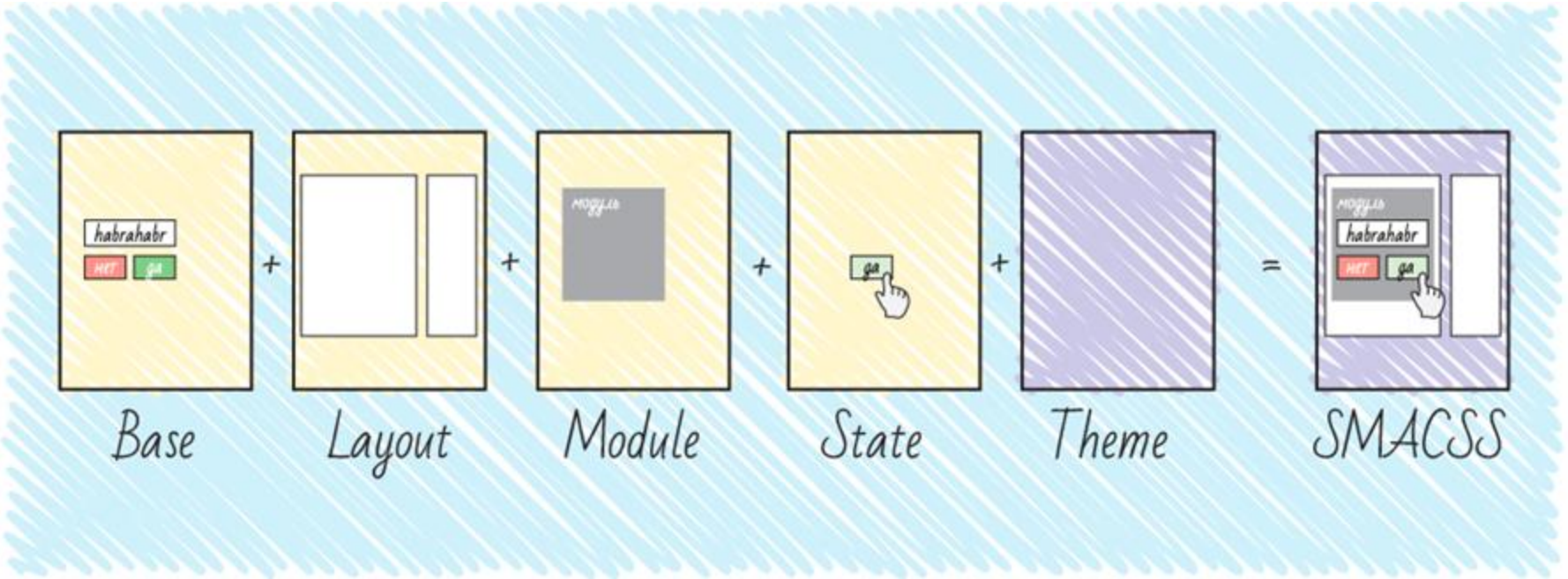
Разделение контейнера и содержимого

Принцип: внешний вид элемента не зависит от того, где он расположен.
Вместо **.my-element button** создаем отдельный стиль **.control** для конкретного случая
Зачем?

- Все **button** будут выглядеть одинаково
- К любому элементу можно добавить класс **.control**. Работает как mixin

.my-element button не нужно переопределять, если передумали

Масштабируемая и модульная архитектура CSS



Выводы

- Избавляемся от каскадирования!
- Придумываем слои: компонент -> приложение -> глобальные стили
- Держим стили внутри своего слоя: **button.html** + **button.css** + **button.js**
- Расширяем компоненты через глобальные mixin-классы: **.animated**, **.themed**, **.hidden**, **.row**

Проблемы:

- Коллизия стилей

CSS через JS

Идея: представить стили через объект, записывать их в тег style, когда посчитаем нужным

Но зачем???

- CSS гибкий, JS ещё гибче — можем делать всё, что захотим
- Темизация страницы в **runtime**
- Уменьшение размера загружаемых стилей => ускорение первого рендера
- МИНУС - Не кешируется

CSS Modules (использование)

```css
/* button.css */
.button {
    width: 200px;
    height: 48px;
    border-radius: 12px;
}

.primary {
    background-color: green;
    font-weight: 500;
}

```

```js
// button.js
import styles from './button.css';

export default function renderButton (title, primary) {
    return `
        <button class="${styles.button} ${primary ? styles.primary : ''}">
            ${title}
        </button>
    `;
}

```

```html
<!-- результирующий HTML -->

<button class="button-213ge1hw primary-jh4gd318">
    Вжух!
</button>
<button class="button-213ge1hw">
    Очистить
</button>

```

JSS (использование)

```
// ...
const { classes } = jss.createStyleSheet(styles).attach();

document.body.innerHTML = `
    <button class="${classes.button}">
        Button
    </button>
`;

```

```
// main.js
import jss from 'jss';
import preset from 'jss-preset-default';
import color from 'color';

// One time setup with default plugins and settings
jss.setup(preset());
const styles = {
    button: {
        width: 200,
        background: color('blue').darken(0.3).hex(),
    },
};

```