Лекция 4

Архитектура Web

“Пользователь вводит ссылку в браузер…”

Подробнее тут https://vc.ru/selectel/76371-chto-proishodit-kogda-polzovatel-nabiraet-v-brauzere-adres-sayta

Поиск сервера

Работу сайта обеспечивает сервер. У сервера есть адрес - **IP-адрес.**

Откуда браузеру взять IP-адрес сервера?

Такая информация хранится в распределенной системе серверов — **DNS** (**Domain Name System**). Система работает как общая «контактная книга», хранящаяся на распределенных серверах и устройствах в интернете.

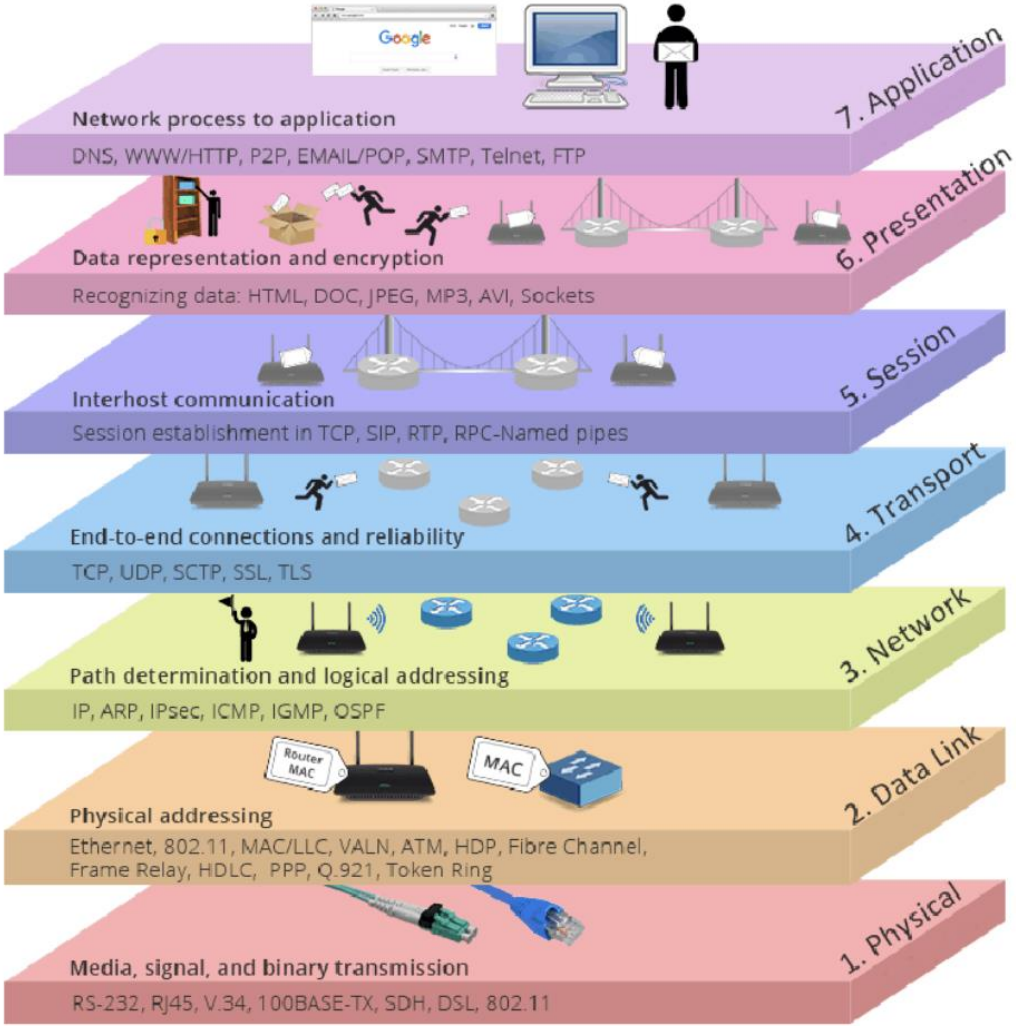
DNS-сервер


.
.org
.ru
.com
.io
apple.com
vc.ru
vscale.io
ya.ru
mail.vc.ru
store.apple.com
taxi.ya.ru

TCP соединение

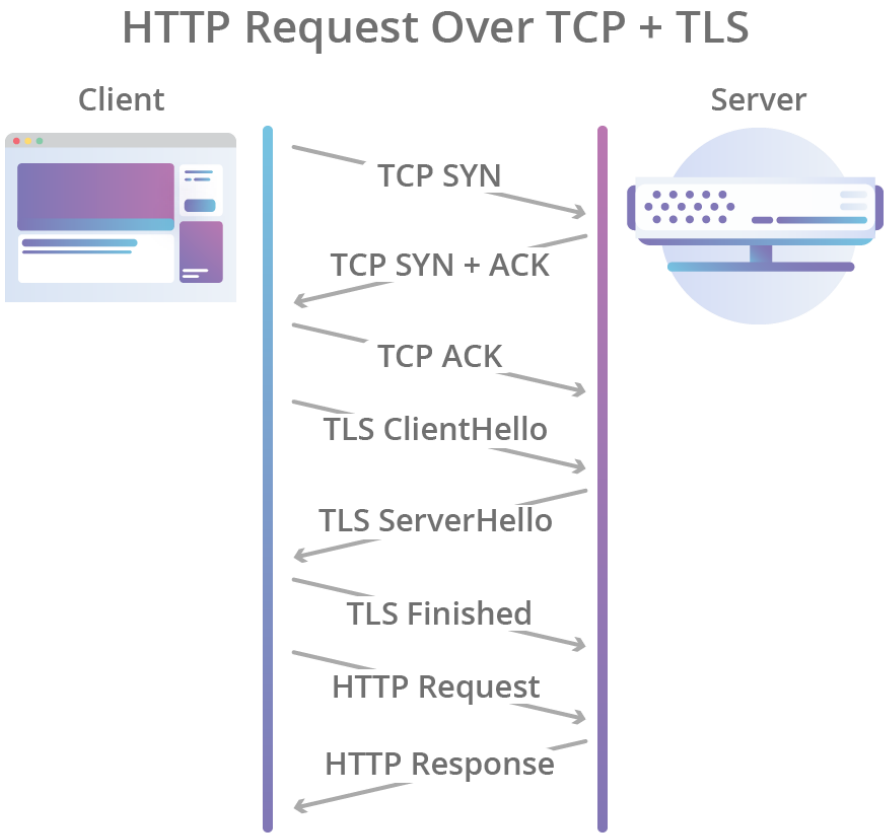
Как только браузер узнал IP-адрес нужного сервера, он пытается установить с ним соединение. В большинстве случаев для этого используется специальный протокол — TCP.

Для установления TCP соединения используется система трех рукопожатий.



Тройное рукопожатие

- Устройство пользователя отправляет специальный запрос на установку соединения с сервером — называется SYN-пакет.
- Сервер в ответ отправляет запрос с подтверждением получения SYN-пакета — называется SYN/ACK-пакет.
- В конце устройство пользователя при получении SYN/ACK-пакета отправляет пакет с подтверждением — ACK-пакет. В этот момент соединение считается установленным.



Остальные шаги

1. Браузер отправляет HTTP-запрос по установленному соединению;
2. Сервер получает запрос и обрабатывает его (бизнес-логика, рендер, проксирование и тд);
3. Сервер отправляет ответ браузера. Это может быть любой (!) ресурс;
4. Браузер обрабатывает полученный ответ и «рисует» веб-страницу;

Как происходит рендер (кратко)

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

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

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.

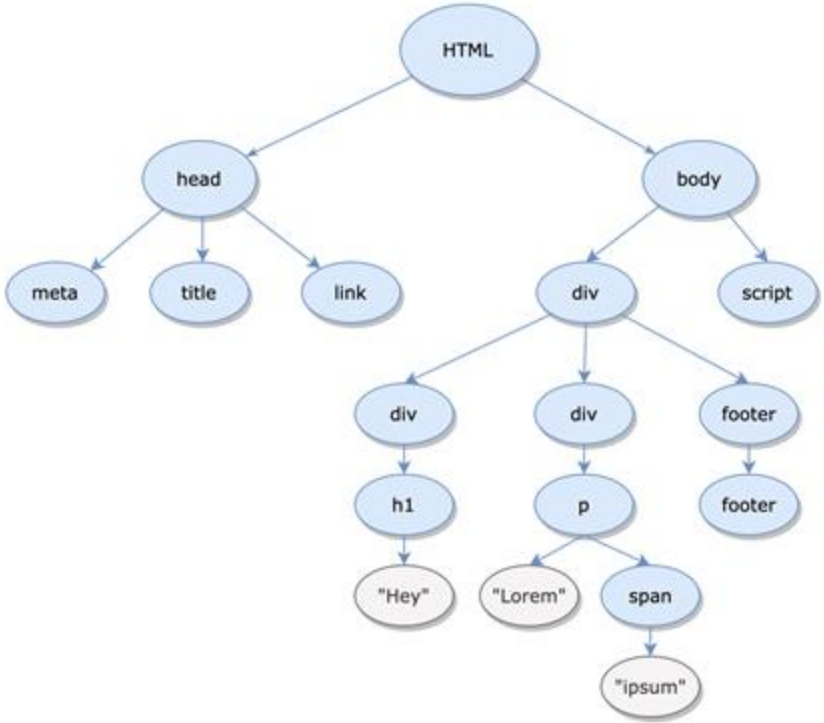
1. Загрузка HTML;
2. DOM;
3. CSSOM;
4. Дерево рендера (render tree);
5. Компоновка (layout);
6. Отрисовка (paint).

Критические этапы рендера (базово)

Браузер парсит загружаемый HTML, преобразуя полученные байты документа в DOM-дерево. Браузер создает новый запрос каждый раз, когда он находит ссылки на внешние ресурсы, будь то файлы стилей, скриптов или ссылки на изображения.
Браузер продолжает парсить HTML и создавать DOM до тех пор, пока запрос на получение HTML не подходит к концу.
После завершения парсинга DOM, браузер конструирует CSS модель.
Как только эти модели сформированы, браузер строит дерево рендера (render tree), в котором вычисляет стили для каждого видимого элемента страницы.
После формирования дерева происходит компоновка (layout), которая определяет положение и размеры элементов этого дерева.
Как только этап завершён - страница рендерится. Или "отрисовывается" (paint) на экране.

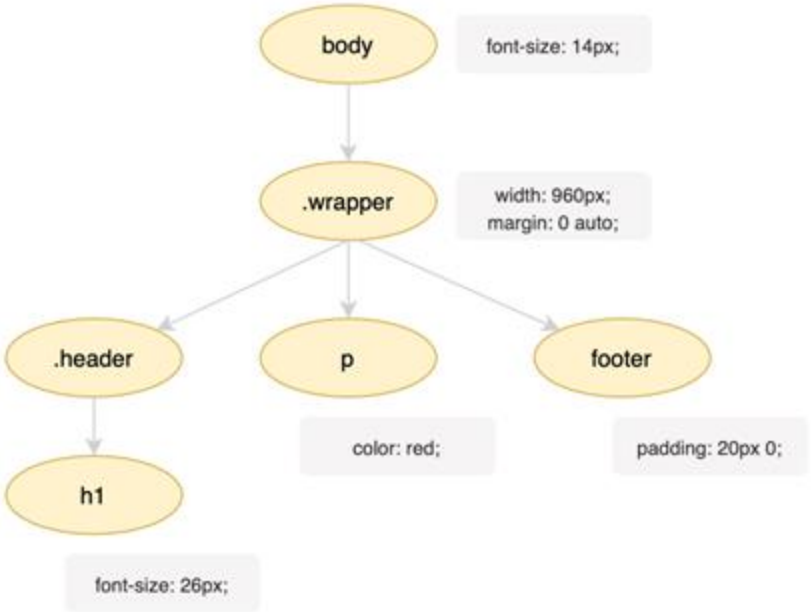
Document Object Model

Построение DOM инкрементально. 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 какой-то набор startTag и endTag-токенов появляется между уже существующим набором токенов, мы получаем узел (node) внутри узла (node), то есть получаем иерархию дерева DOM.



CSS Object Model

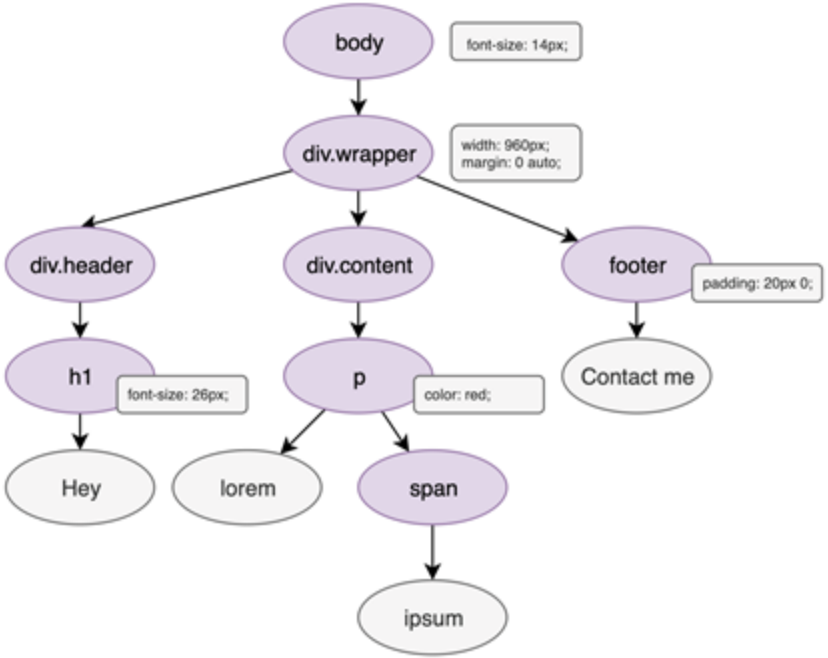
DOM несёт в себе всё содержимое страницы. CSSOM содержит все стили страницы, то есть данные о том, как стилизовать DOM. CSSOM похож на DOM, но всё же отличается. Если формирование DOM инкрементально, CSSOM - нет. CSS блокирует рендер: браузер блокирует рендеринг страницы до тех пор, пока не получит и не обработает все CSS-правила. CSS блокирует рендеринг, потому что правила могут быть перезаписаны, а значит, 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, то есть размеру viewport'а.

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

Подводя промежуточный итог: **браузер знает что, как и где рисовать**. Следовательно — 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, так как браузер старается оптимизировать процесс отрисовки, избегая ненужной работы. Так, если у вас в документе есть два элемента, перерисовываться будет только тот, который вы изменили.

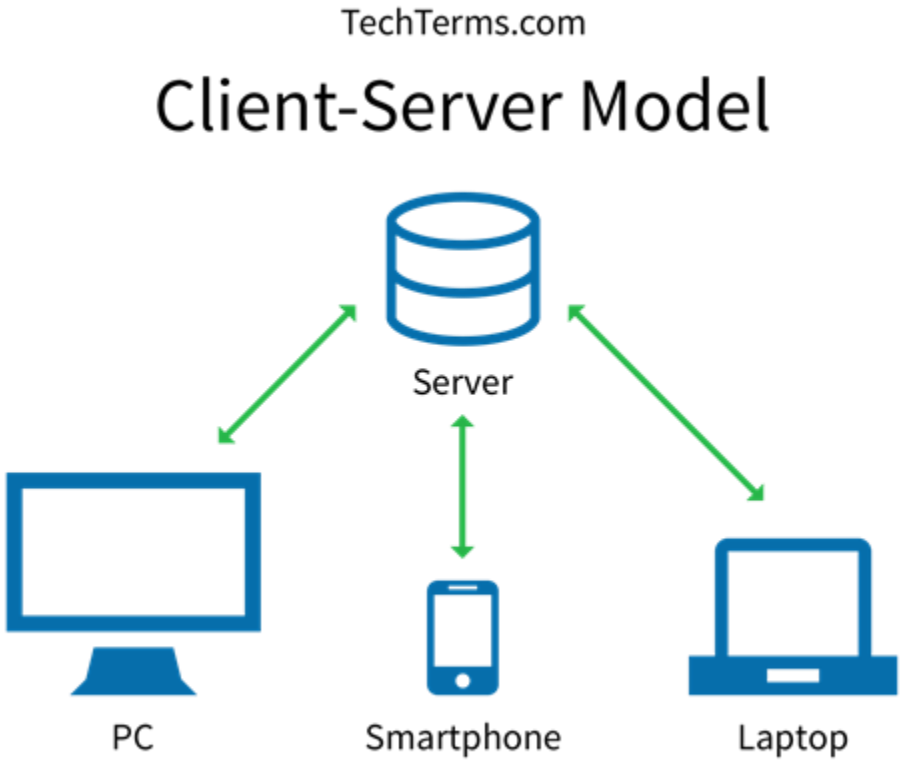
Клиент-серверная модель

Подробнее тут: https://habr.com/ru/post/495698/

Клиент-сервер

Клиент – серверная модель — вычислительная или сетевая архитектура, в которой задания или сетевая нагрузка распределены между поставщиками услуг, называемыми серверами, и заказчиками услуг, называемыми клиентами

Клиент-сервер



История веб-приложения

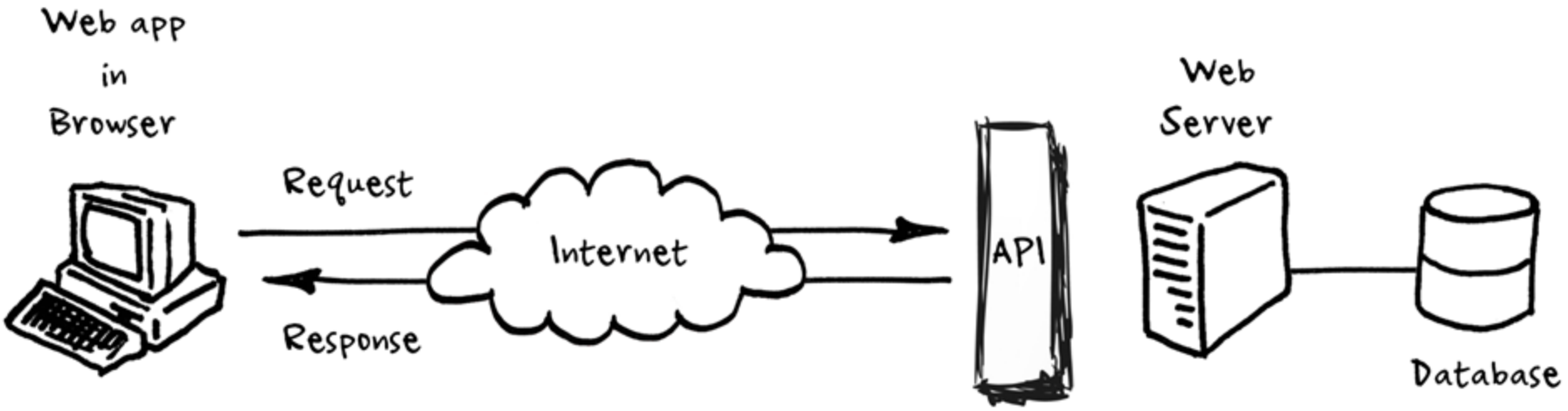
Подробнее тут: нигде

Статические веб-страницы

Статические веб-страницы

- Frontend
  a. Отображение статических HTML-страниц (HTML, CSS);
  b. Контент на страницах, переходы по гиперссылкам
- Backend
  a. Хранение статических документов и отдача по запросу по протоколу HTTP

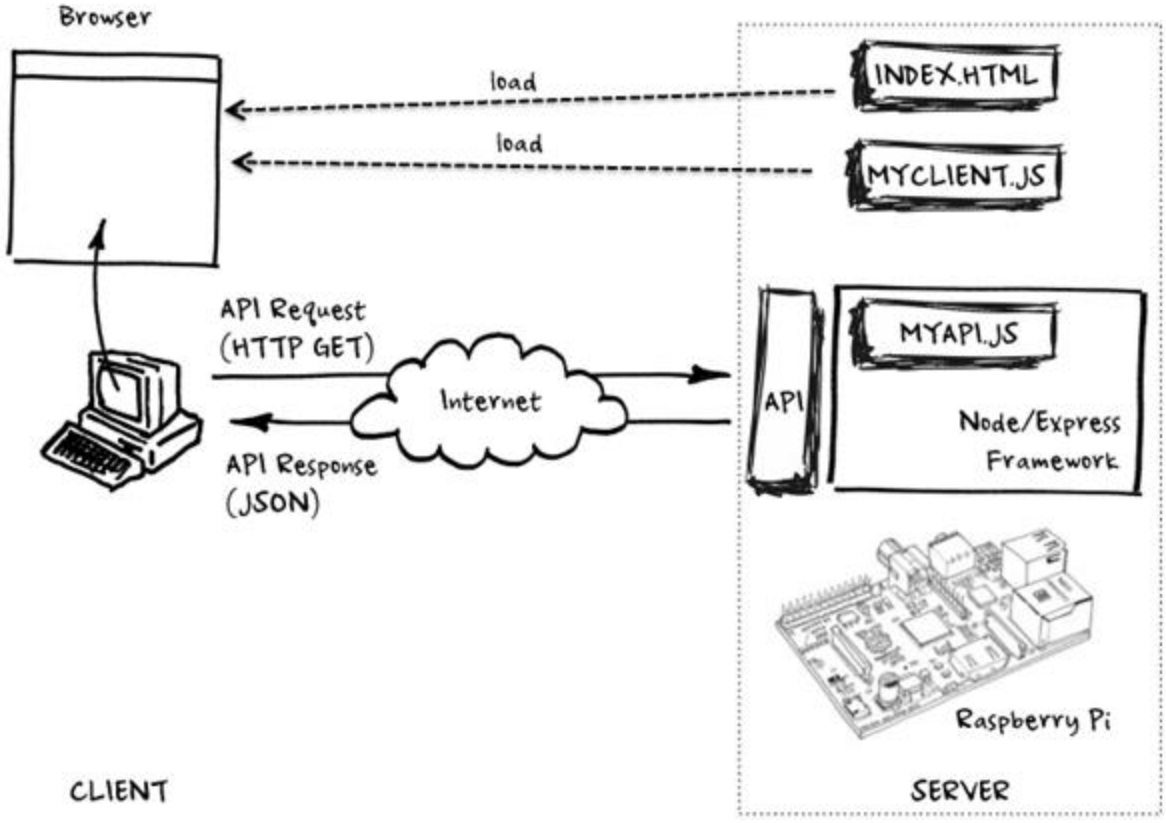
Динамические веб-сервисы



Динамические веб-сервисы

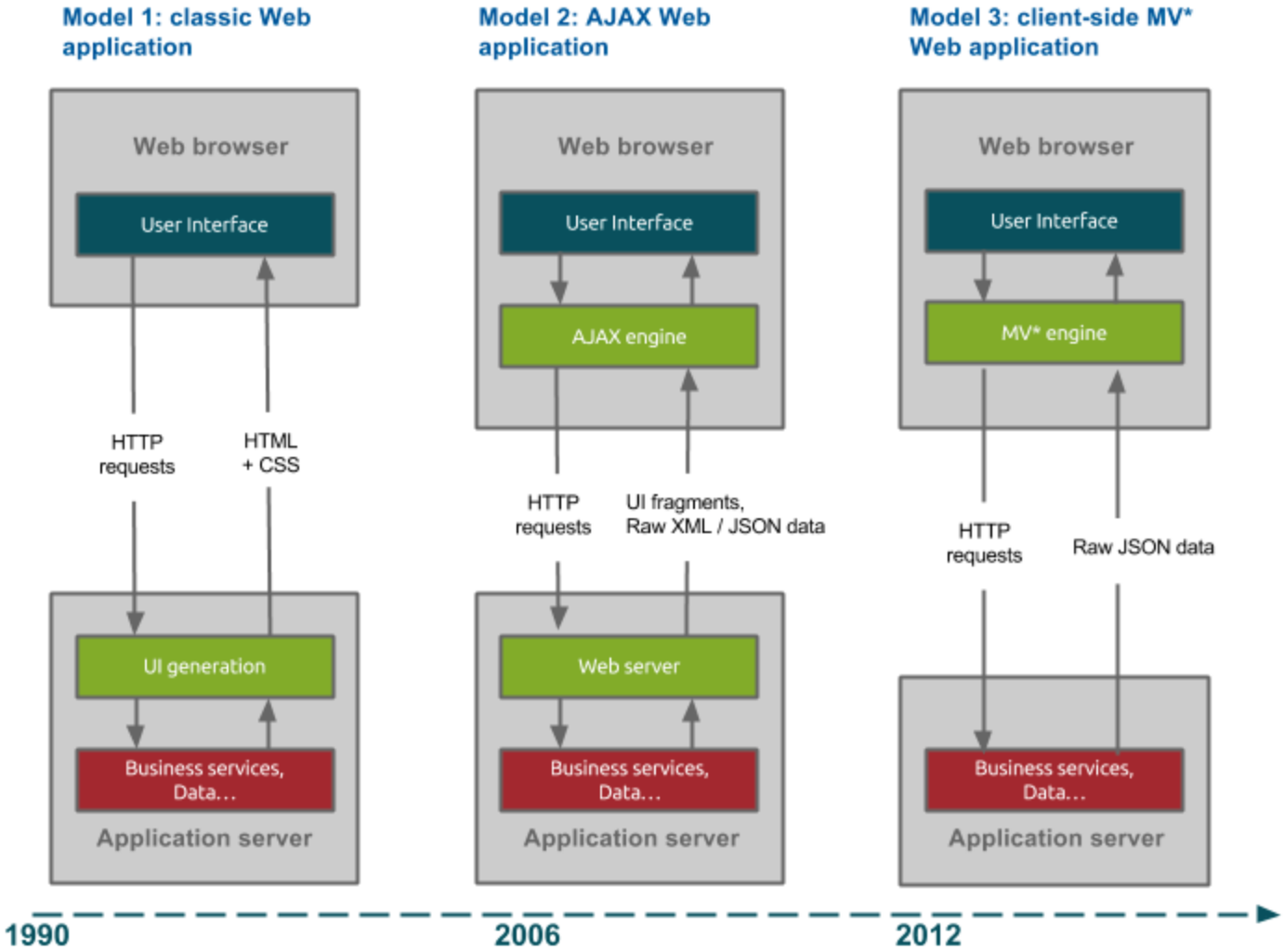
- Frontend
  a. Отображение статических HTML-страниц (HTML, CSS)
  b. Контент на страницах, переходы по гиперссылкам
  c. Взаимодействие с сервисом посредством форм
- Backend
  a. Хранение статических документов и отдача по запросу по протоколу HTTP
  b. Обработка пользовательских запросов и генерация динамических страниц
  c. Хранение данных в базе данных

Веб-приложения



Динамические веб-сервисы

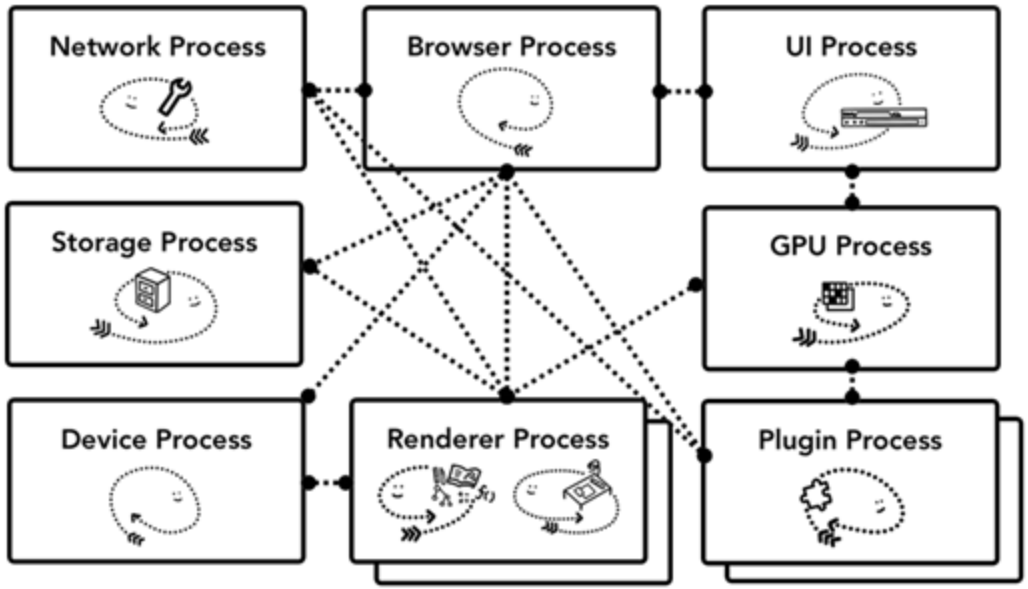
- Frontend
  a. Хранение и доступ к статическому контенту (файлы стилей, скрипты)
  b. Генерация и отображение пользовательского интерфейса
  c. Взаимодействие с пользователем и выполнение запросов к API
  d. Обновление пользовательского интерфейса в ответ на действия пользователя
- Backend
  a. Реализация публичного API
  b. Хранение данных в базе данных и работа с ними


Model 1: classic Web application
Web browser
User Interface
HTTP requests
HTML + CSS
UI generation
Business services, Data…
Application server
Model 2: AJAX Web application
Web browser
User Interface
AJAX engine
HTTP requests
UI fragments, Raw XML / JSON data
Web server
Business services, Data…
Application server
Model 3: client-side MV* Web application
Web browser
User Interface
MV* engine
HTTP requests
Raw JSON data
Business services, Data…
Application server
1990
2006
2012

Node && Npm

Подробнее тут https://nodejs.dev/

Что находится внутри браузера?



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)

Node.js

Node.js - исполнение JavaScript кода НЕ в браузере.

Программная платформа, основанная на движке V8, превращающая JavaScript из узкоспециализированного языка в язык общего назначения. Node.js добавляет возможность JavaScript взаимодействовать с устройствами ввода-вывода через свой API, написанный на C++, подключать другие внешние библиотеки, написанные на разных языках, обеспечивая вызовы к ним из JavaScript-кода.

Npm

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

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

- Вариант с Npm

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

Npm (как работает)

- Скачиваем нужные нам зависимости

```
1 npm install bootstrap
```

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

```
1 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

Npm (создаем свой модуль)

```
{
  "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"
  }
}

```

Архитектура

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

Архитектура

Если рассматривать приложение как систему — т.е. набор компонентов, объединенных для выполнения определенной функции:

- ***Архитектура*** *идентифицирует главные компоненты системы и способы их взаимодействия. Также это выбор таких решений, которые интерпретируются как основополагающие и не подлежащие изменению в будущем*
- ***Архитектура*** *— это организация системы, воплощенная в её компонентах, их отношениях между собой и с окружением.*

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

Проектирование системы

Критерии *хорошего* дизайна системы

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

Проектирование системы

Критерии *плохого* дизайна системы

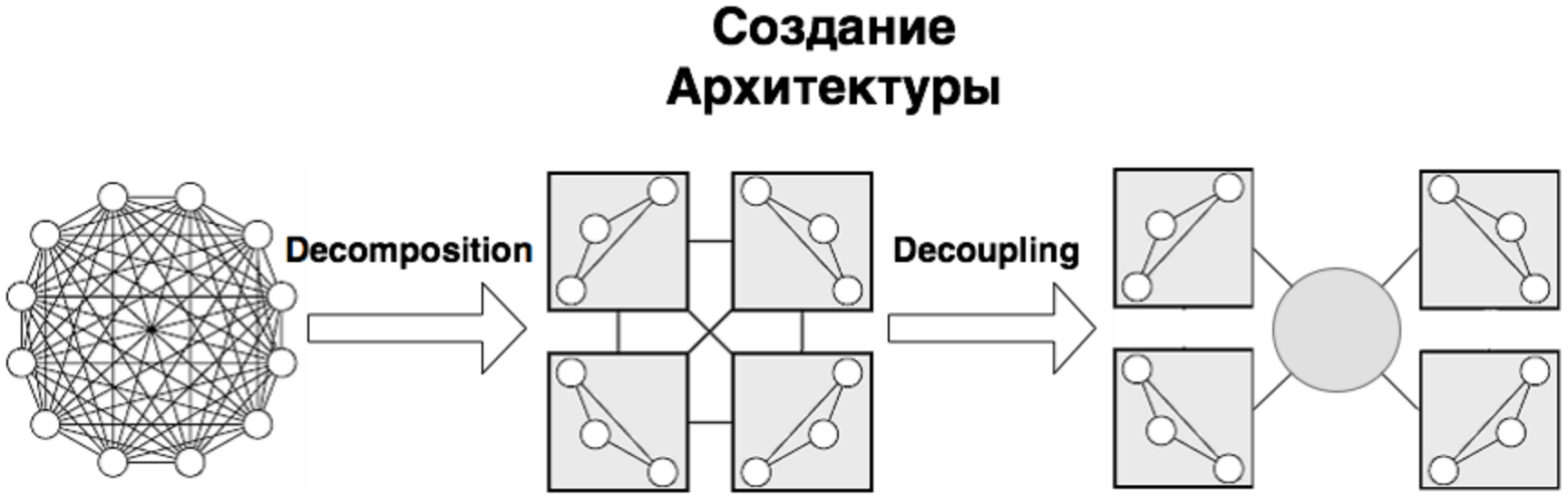
- Его тяжело изменить, поскольку любое изменение влияет на слишком большое количество других частей системы
- При внесении изменений неожиданно ломаются другие части системы
- Код тяжело использовать повторно в другом приложении, поскольку его слишком тяжело «выпутать» из текущего приложения

Методологии и принципы

- **DRY** — don’t repeat yourself (не повторяйте себя)
- **KISS** — keep it simple stupid (делайте вещи проще)
- **YAGNI** — you ain’t gonna need it (вам это не понадобится)
- **GRASP** — документированные и стандартизированные принципы объектно-ориентированного анализа
- **S.O.L.I.D.** — пять основных принципов объектно-ориентированного программирования и проектирования
- **PACKAGE PRINCIPLES** — package cohesion (REP, CRP и CCP) и package coupling (ADP, SDP и SAP)

Для создания хорошей архитектуры используем проверенные подходы

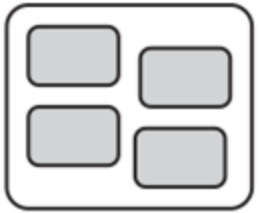
Декомпозиция



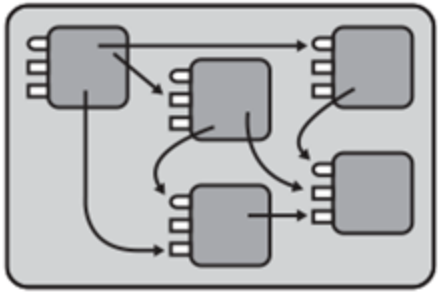
Хорошая декомпозиция - иерархическая

Декомпозиция (иерархическая)

① Программная система



② Разделение системы на подсистемы/пакеты



③ Разделение пакетов на классы

Хорошая декомпозиция - функциональная

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 }
```

Декомпозиция (функциональная)

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

Декомпозиция (советы)

- **ОДИН файл — ОДИН модуль** — самое главное правило, если появляется необходимость определить в одном файле несколько модулей, значит вы что-то делаете не так
- Не допускать *циклические* зависимости — есть специальные алгоритмы избавления от них
- Не создавайте *утилитарные* мега-модули с множеством экспортируемых функций — создавайте просто набор маленьких функций-модулей

Хорошая декомпозиция создает слабосвязанную систему

Декомпозиция (связанность)

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

Декомпозиция (связанность)

Модули, полученные в результате декомпозиции, должны быть максимально сопряженны внутри (**high internal cohesion**) и минимально связанны друг с другом (**low external coupling**)

Модули, на которые разбивается система, должны быть, по возможности, независимы или слабо связанны друг с другом. Они должны иметь возможность взаимодействовать, но при этом как можно меньше знать друг о друге

Декомпозиция (ослабляем связанность)

- 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?

Декомпозиция (способы снижения связанности)

- Использование паттернов ООП, интерфейсы, фасад
- Использование Dependency Inversion — корректное создание и получение зависимостей
- Замена прямых зависимостей на обмен сообщениями
- Замена прямых зависимостей на синхронизацию через общее ядро
- Следование Закону Деметры (law of Demeter), запрет неявных и транзитивных зависимостей
- Композиция вместо наследования

Observable

```
1 // on-off
2 class SomeService {
3     constructor () { ... }
4     do () { ... }
5
6     on(event, callback) { ... }
7     off(event, callback) { ... }
8     emit(eventName, eventData) { ... }
9 }
```

Observable

```
const service = new SomeService();
// функция-обработчик события
const onload = function (data) { console.log(data); }

// подписываемся на событие
service.on('loaded', onload);
service.emit('loaded', {data: 42});     // событие 1
service.emit('loaded', {foo: 'bar'});   // событие 2
// отписываемся от события
service.off('loaded', onload);
```

Observable (в чем преимущество)

- Все Observable-модули реализуют один и тот же интерфейс методов
- Модули, на которые мы подписываемся, сами знают, когда и какие события эмитить
- Модули, которые подписываются, сами знают, как обрабатывать эти события
- И те, и другие, ничего друг о друге не знают, модулям известны только названия и формат сообщений
- Все необходимые подписки происходят в одном месте (main.js)
- Удобно оповещать систему о новых событиях

Composition (создаем МФУ)

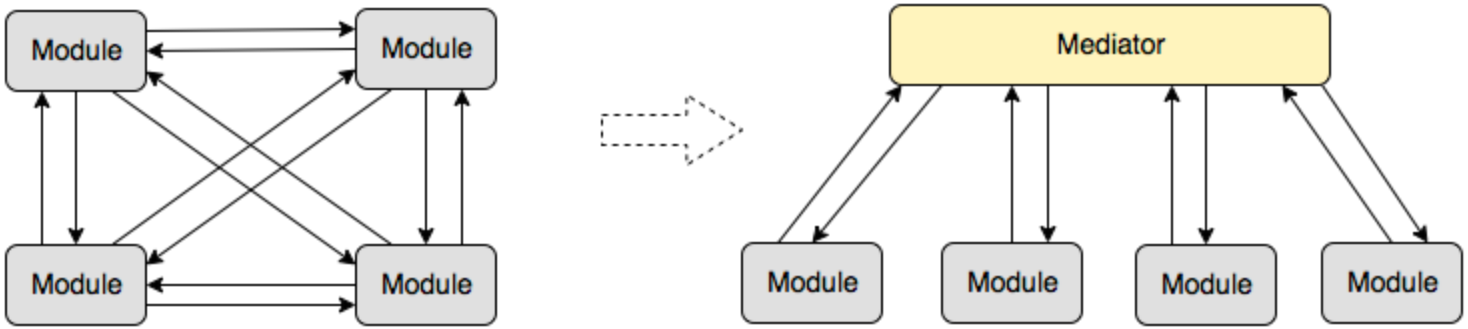
```
1  class ЧБПринтер {
2      print (doc) { ... }
3  }
4
5  class ЦветнойПринтер {
6      print (doc) { ... }
7  }
8
9  class Сканер {
10     scan (doc) { ... }
11 }
```

Composition (создаем МФУ)

```
1 class МФУ {
2     constructor () {
3         this.чбПринтер = new ЧБПринтер();
4         this.цветнойПринтер = new ЦветнойПринтер();
5         this.сканер = new Сканер();
6         // ...
7     }
8
9     scan (doc) {
10        return this.сканер.scan(doc);
11    }
12 }
```

Mediator

Когда в системе присутствует большое количество модулей, их прямое взаимодействие друг с другом (даже с учётом применения publish-subscriber подхода) становится слишком сложным. Поэтому имеет смысл взаимодействие «все со всеми» заменить на взаимодействие **«один со всеми»**. Для этого вводится некий обобщенный посредник — **медиатор**



Mediator

```
1 // modules/event-bus.js
2 class EventBus {
3     on(event, callback) { ... }
4     off(event, callback) { ... }
5     emit(eventName, eventData) { ... }
6 }
7
8 export default new EventBus();
```

Mediator

```
1  // blocks/user-profile.js
2  import bus from '../modules/event-bus.js';
3
4  export default class UserProfile {
5      constructor () {
6          bus.on('user:logged-in', function (user) {
7              // do stuff
8
9              this.render();
10         }.bind(this))
11     }
12 }
```

Декомпозиция (советы)

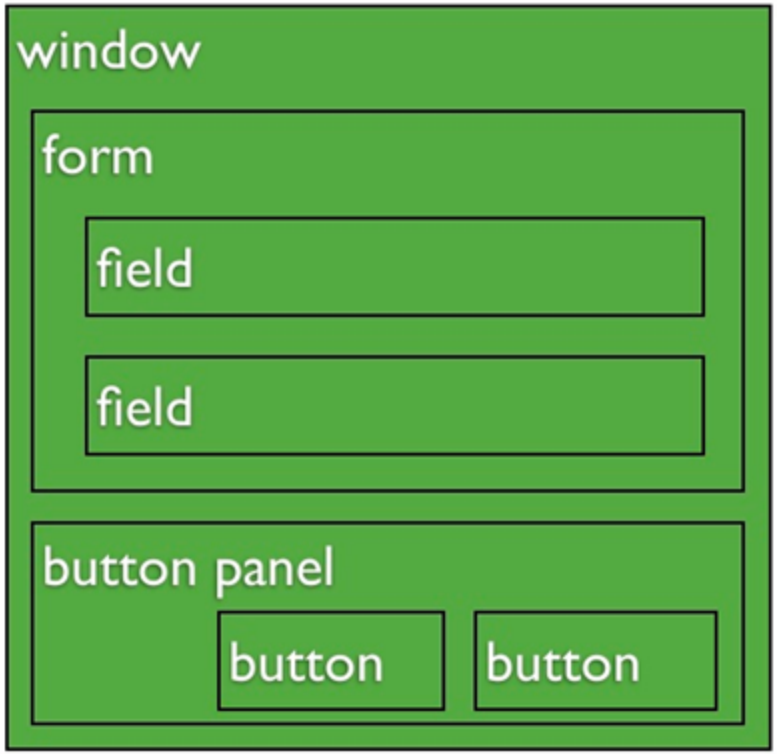
- Минимизировать использование глобальных зависимостей
- Не использовать неявные зависимости — все зависимости модуля должны быть обозначены явно
- Минимизировать связи между модулями — например, с помощью паттернов Observable и Mediator

Хорошая декомпозиция - компонентная

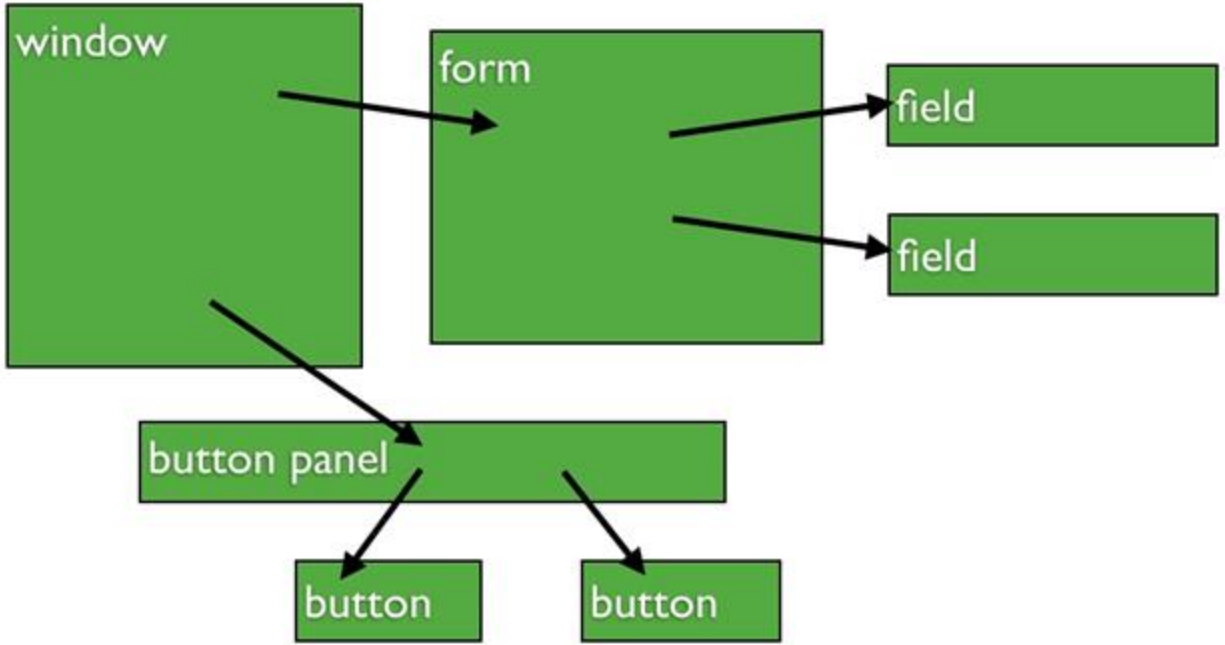
Декомпозиция (компоненты)

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

Обычный подход



Компонентный подход



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

Архитектура JS

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

Как нам правильно организовать код?

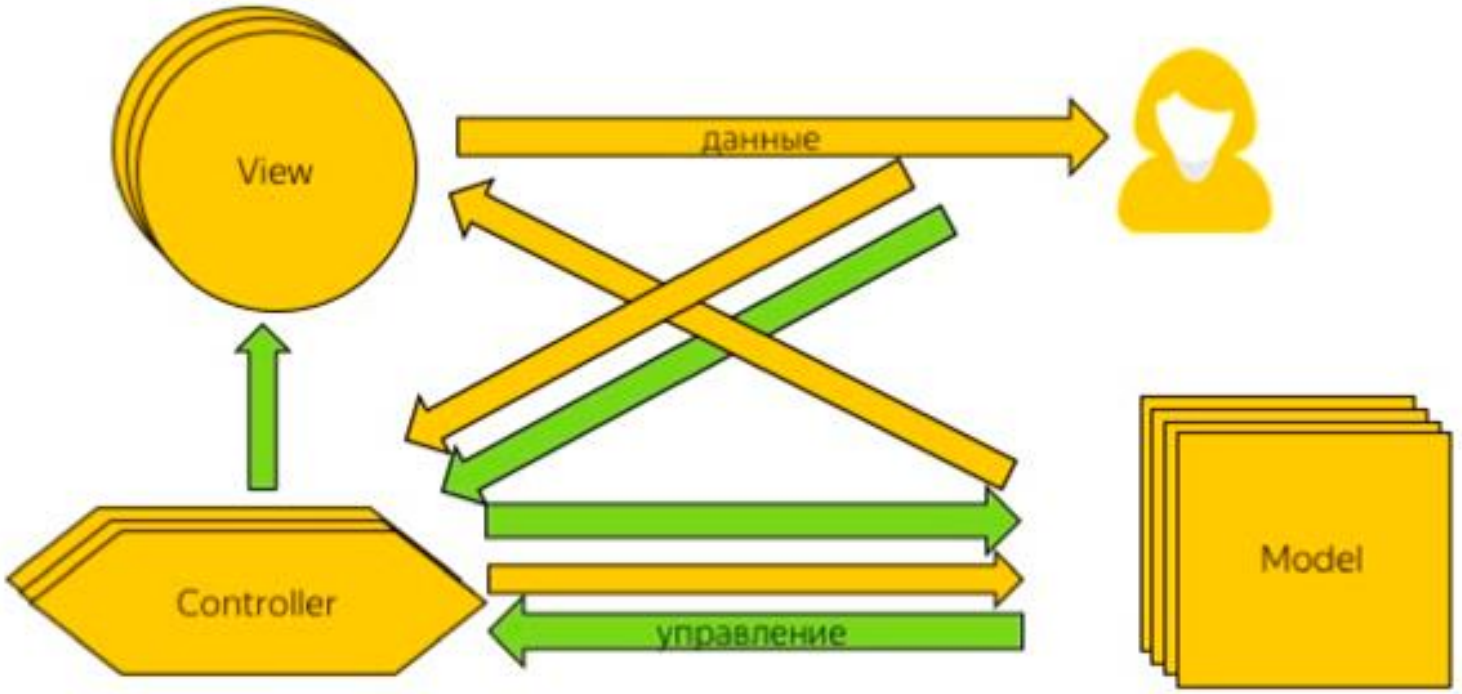
Шаблоны MVC

MV?

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

Шаблоны MVC



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

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

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 все вместе

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**

Собираем MVC

- У нас уже имеются сервисы и модули, которые выполняют роли моделей в нашем приложении и описывают бизнес-логику работы с данными
- Настраиваем взаимодействие между частями приложения посредством контроллеров (например, медиатора)
- Во время работы приложения управление передаётся между разными **View**
- Стараемся уменьшить связность приложения, отделив модели и представления друг от друга (проведя правильную декомпозицию)

Итоги MVC

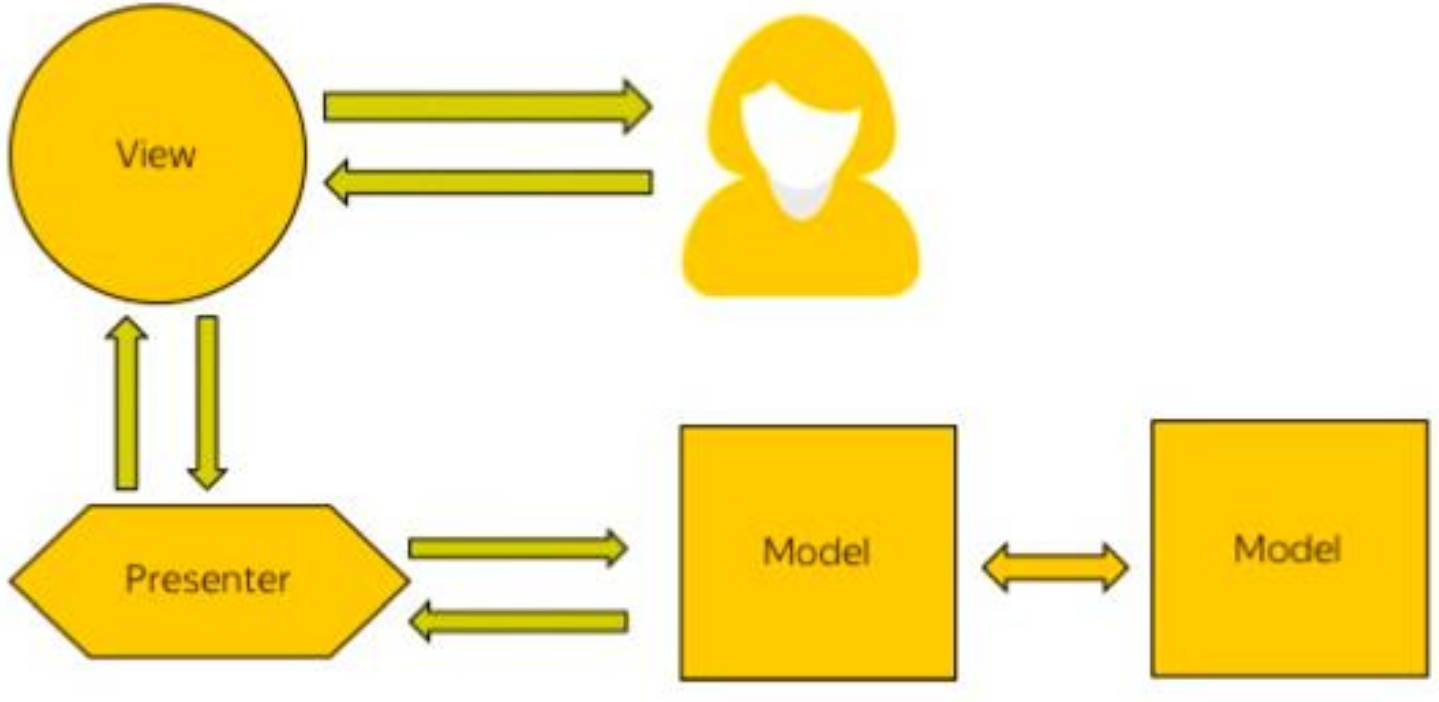
Пример: заполнение формы и отправка на сервер

- Controller и Model находятся на бэкенде, серверная шаблонизация View

Проблемы:

- Не подходит для SPA и активного ajax

Шаблоны MVP



Итоги MVP

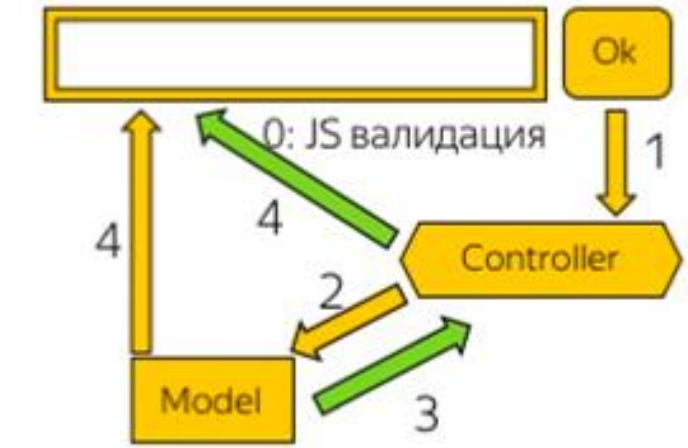
Пример: заполнение формы и отправка на сервер

- Model может быть на бекенде или клиенте, Controller и View на клиенте

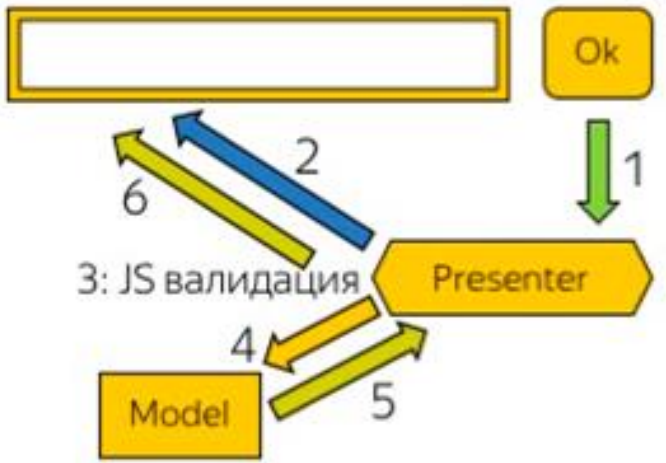
Проблемы:

- Не подходит для SPA и активного ajax (проблемы нет)

Пример MVC/MVP



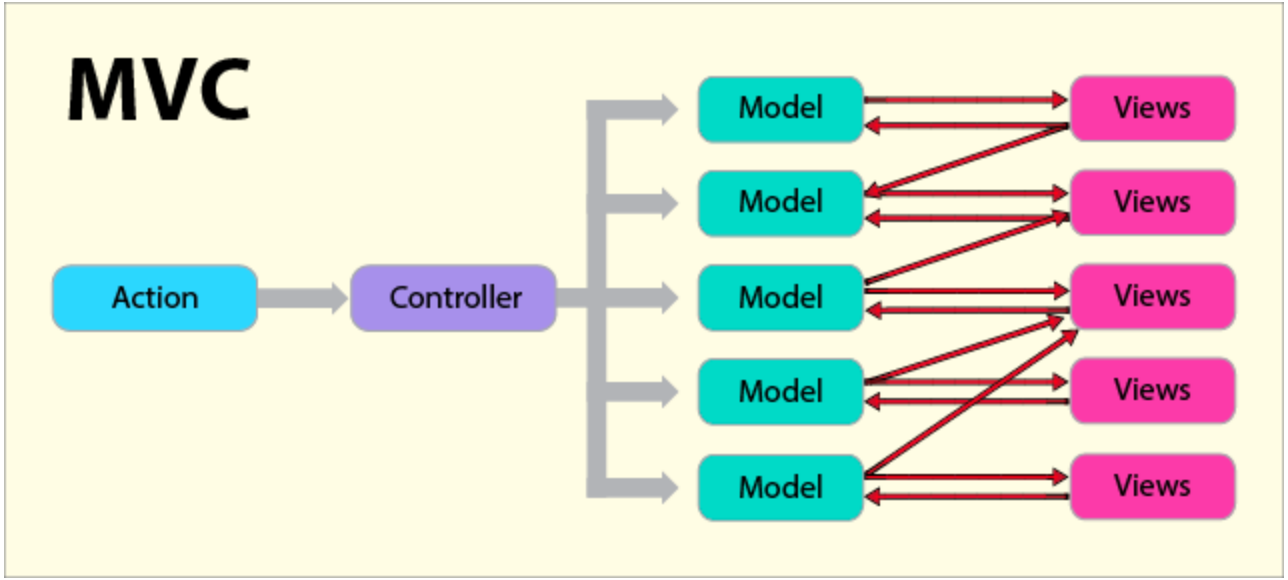
MVC



MVP

MVP решает
все проблемы ?

Hea


MVC
Action
Controller
Model
Model
Model
Model
Model
Views
Views
Views
Views
Views

Flux

Flux-архитектура — архитектурный подход или набор шаблонов программирования для построения пользовательского интерфейса веб-приложений, сочетающийся с реактивным программированием и построенный на однонаправленных потоках данных.

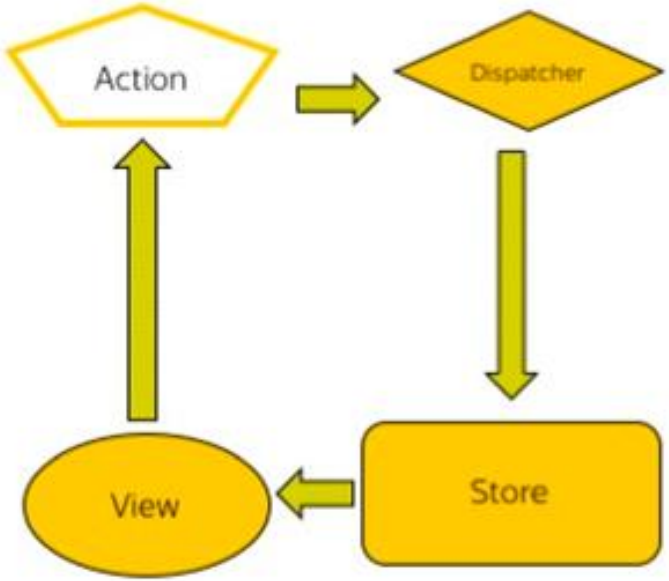
Основной отличительной особенностью Flux является односторонняя направленность передачи данных между компонентами Flux-архитектуры. Архитектура накладывает ограничения на поток данных, в частности, исключая возможность обновления состояния компонентов самими собой. Такой подход делает поток данных предсказуемым и позволяет легче проследить причины возможных ошибок в программном обеспечении

Flux (основные понятия)

- **Actions** (действия) - событие (объект), который совершил пользователь
- **Dispatcher** (диспетчер) - передает действие хранилищу
- **Stores** (хранилища) - хранит данные пользователя
- **Views** (представления) - показывает данные пользователю

Flux

1. **View**: создает *структуру* типа **Action**, передает ее в **Dispatcher**
2. **Dispatcher** вызывает *коллбэк* из **Store**
3. **Store**:
   - смотрит на *метаданные* **Action**-а, выбирает метод обновления
   - Обновляет данные
   - Триггерит события *изменения*
4. **View**: *берет* данные из **Store** и *перерисовывается*



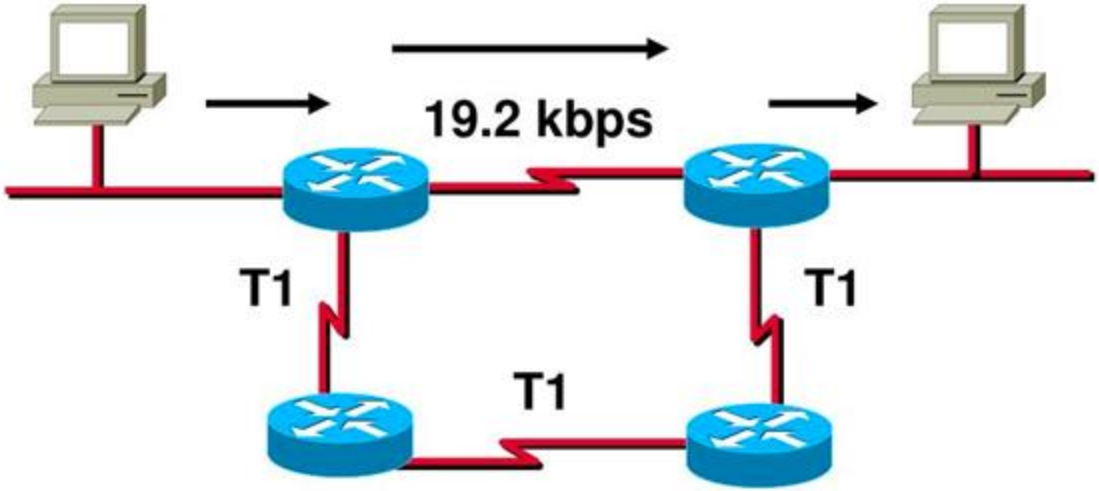
Пример Flux


6: перерисовка
Ok
View
1: создание action-a
4: событие
'change'
5: запрос
данных
Action
{
type: 'changeName',
data: 'Вася'
}
0: регистрация
callback-a
1: вызов метода
3: Валидация
Обновление
Store
Dispatcher
2: вызов
callback-a

Роутинг

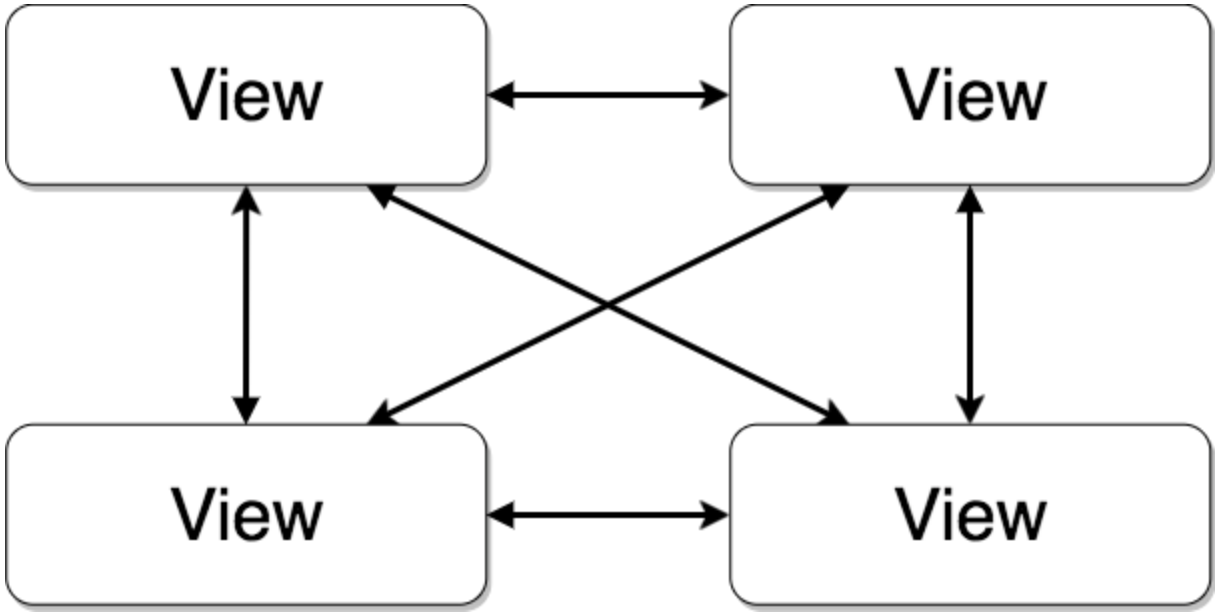
Подробнее тут https://frontend.tech-mail.ru/slides/s5/

Routing Protocol (RIP)



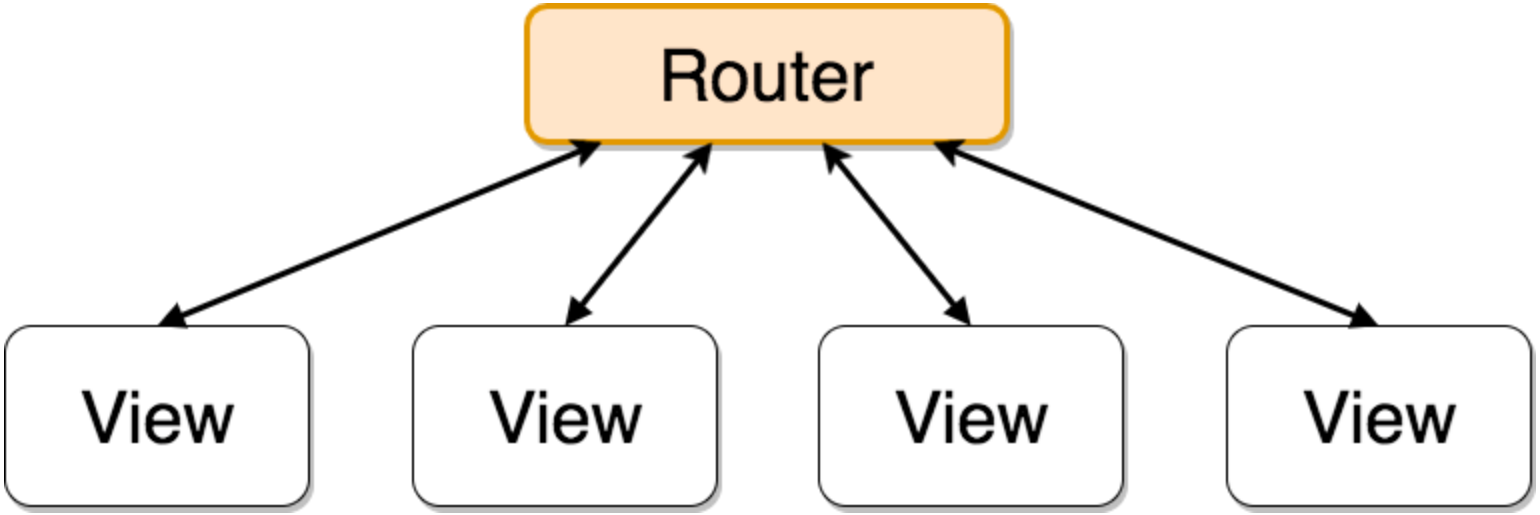
- Hop count metric selects the path
- Routes update every 30 seconds

Вернемся назад



Вспоминаем про медиатор

Роутер



Что такое роутинг?

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

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

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({state: 'main'}, MenuView);
2 Router.register({state: 'signup'}, SignupView);
3 Router.register({state: 'scores'}, ScoreboardView);
4 ...
5 // переход на 3 страницу (пагинация)
6 Router.go({state: 'scores', params: {page: 3}});
7
```

History API

History API — браузерное API, позволяет манипулировать историей браузера в пределах сессии , а именно историей о посещённых страницах в пределах вкладки или фрейма, загруженного внутри страницы. Позволяет перемещаться по истории переходов, а так же управлять содержимым адресной строки браузера

History API

```
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
```

History API

```
1 // Изменение истории
2 const state = { foo: 'bar' };
3 window.history.pushState(
4     state,          // объект состояния
5     'Page Title',   // заголовок состояния
6     '/pages/menu'   // URL новой записи (same origin)
7 );
8 window.history.replaceState(state2, 'Other Title', '/another/page');
9
```

Что такое роутинг?

```
1 class Router {
2     constructor() { ... }
3     register(path: string, view: View) { ... }
4     start() { ... }       // запустить роутер
5     go(path: string) { ... }
6     back() { ... }        // переход назад по истории браузера
7     forward() { ... }     // переход вперёд по истории браузера
8 }
9
```

Что такое роутинг?

```
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
```

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

CSS

```
/* 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
```

Какие могут быть проблемы?

```
1 /* Супер классы! */
2 .super-class {
3   margin: 10px;
4   position: absolute;
5   background: black;
6   color: white;
7   transition: color 0.2s;
8   .....
9 }
10
```

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

OOCSS

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

Объектно-ориентированный CSS

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

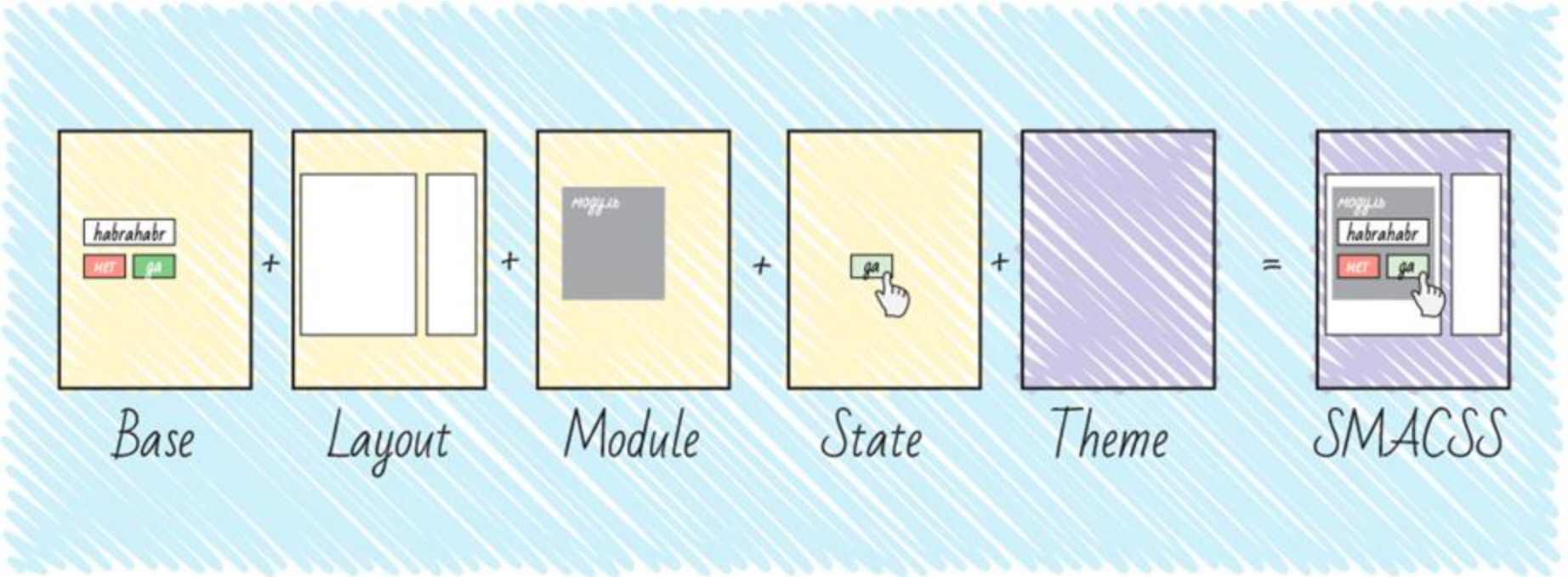
Зачем?

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

.my-element button не нужно переопределять, если передумали

SMACSS

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-in-JS

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
- Не кешируется :(

CSS Modules

CSS Modules (использование)

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

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

```

CSS Modules (использование)

```
1  // button.js
2  import styles from './button.css';
3
4  export default function renderButton (title, primary) {
5      return `
6          <button class="${styles.button} ${primary ? styles.primary : ''}">
7              ${title}
8          </button>
9      `;
10 }
11
```

CSS Modules (использование)

```
1 <!-- результирующий HTML -->
2
3 <button class="button-213ge1hw primary-jh4gd318">
4     Вжух!
5 </button>
6 <button class="button-213ge1hw">
7     Очистить
8 </button>
9
```

JSS

JSS (использование)

```
1  // main.js
2  import jss from 'jss';
3  import preset from 'jss-preset-default';
4  import color from 'color';
5
6  // One time setup with default plugins and settings
7  jss.setup(preset());
8  const styles = {
9      button: {
10         width: 200,
11         background: color('blue').darken(0.3).hex(),
12     },
13 };
14
```

JSS (использование)

```
1 // ...
2 const { classes } = jss.createStyleSheet(styles).attach();
3
4 document.body.innerHTML = `
5     <button class="${classes.button}">
6         Button
7     </button>
8 `;
9
```