

Лекция 5

Асинхронный JavaScript и всякие приколы

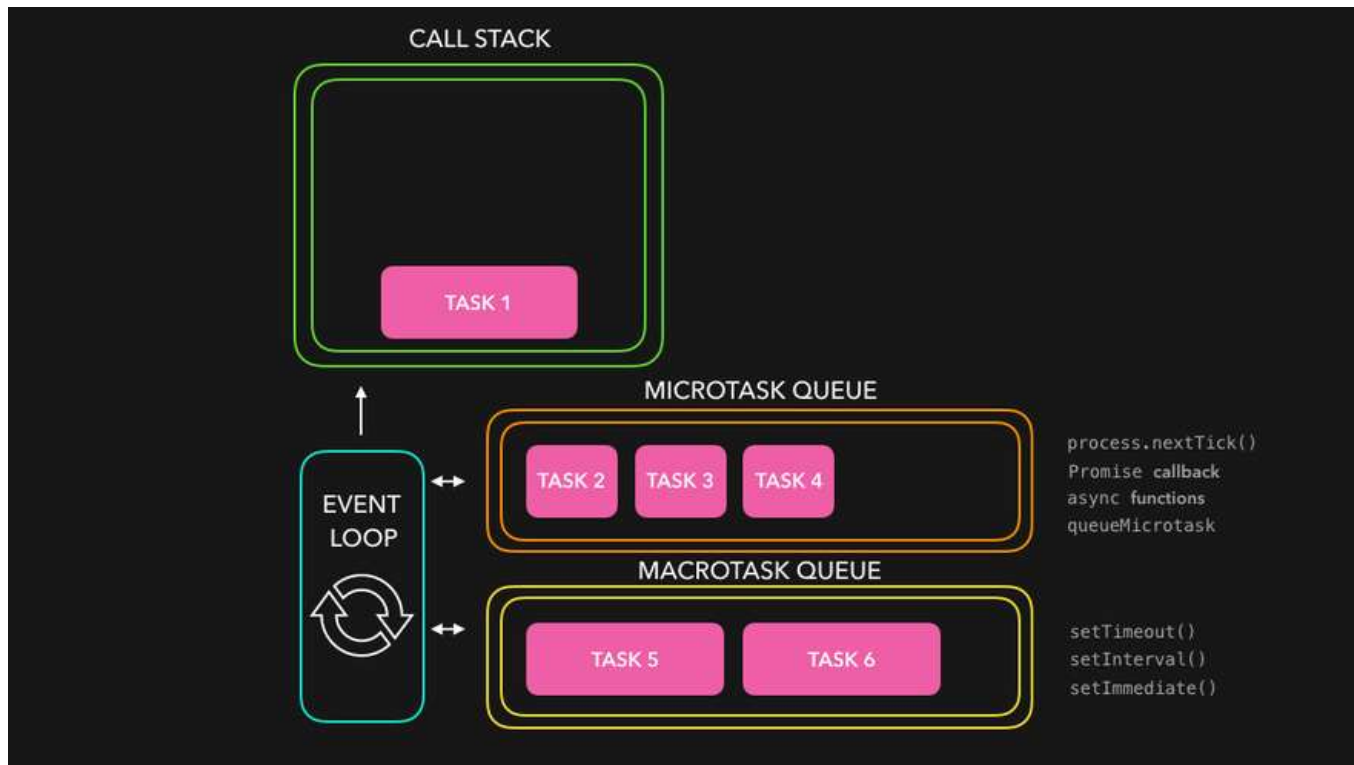
Асинхронность JS

Подробнее тут <https://learn.javascript.ru/callbacks>

Однопоточный и асинхронный



Дядя Event Loop



Так в чем же заключается асинхронность?

```
1 function loadScript(src) {  
2   let script = document.createElement('script');  
3   script.src = src;  
4   document.head.append(script);  
5 }  
6  
7 loadScript('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js');  
8 alert('Ура, скрипт загрузился! (на самом деле нет)');  
9
```

Так в чем же заключается асинхронность?



```
1 function loadScript(src, callback) {  
2   let script = document.createElement('script');  
3   script.src = src;  
4   script.onload = () => callback(script);  
5   document.head.append(script);  
6 }  
7  
8 loadScript('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js', script => {  
9   alert(`Здорово, скрипт ${script.src} загрузился`);  
10  alert( _ ); // функция, объявленная в загруженном скрипте  
11 });  
12
```

Тут есть маленькая проблема...

```
1 loadScript('/my/script.js', function(script) {  
2  
3     loadScript('/my/script2.js', function(script) {  
4  
5         loadScript('/my/script3.js', function(script) {  
6             // ...и так далее, пока все скрипты не будут загружены  
7         });  
8  
9     })  
10  
11 }):
```


Callback Hell

```
function register()
{
    if (!empty($_POST)) {
        $msg = '';
        if ($_POST['user_name']) {
            if ($_POST['user_password_new']) {
                if ($_POST['user_password_new'] === $_POST['user_password_repeat']) {
                    if (strlen($_POST['user_password_new']) > 5) {
                        if (strlen($_POST['user_name']) < 65 && strlen($_POST['user_name']) > 1) {
                            if (preg_match('/^[a-z\d]{2,64}$/i', $_POST['user_name'])) {
                                $user = read_user($_POST['user_name']);
                                if (!isset($user['user_name'])) {
                                    if ($_POST['user_email']) {
                                        if (strlen($_POST['user_email']) < 65) {
                                            if (filter_var($_POST['user_email'], FILTER_VALIDATE_EMAIL)) {
                                                create_user();
                                                $_SESSION['msg'] = 'You are now registered so please login';
                                                header('Location: ' . $_SERVER['PHP_SELF']);
                                                exit();
                                            } else $msg = 'You must provide a valid email address';
                                        } else $msg = 'Email must be less than 64 characters';
                                    } else $msg = 'Email cannot be empty';
                                } else $msg = 'Username already exists';
                            } else $msg = 'Username must be only a-z, A-Z, 0-9';
                        } else $msg = 'Username must be between 2 and 64 characters';
                    } else $msg = 'Password must be at least 6 characters';
                } else $msg = 'Passwords do not match';
            } else $msg = 'Empty Password';
        } else $msg = 'Empty Username';
        $_SESSION['msg'] = $msg;
    }
    return register_form();
}
```



Как можно победить?

```
1 function onLoadScript1(script) {
2     // логика...
3     loadScript(onLoadScript2);
4 }
5
6 function onLoadScript2(script) {
7     // логика...
8     loadScript(onLoadScript3);
9 }
10
11 function onLoadScript3(script) {
12     // логика...
13 }
14
15 loadScript('/my/script.js', onLoadScript1);
16
```

Еще одна проблема...

```
1 function loadScript(src, callback) {
2   let script = document.createElement('script');
3   script.src = src;
4   script.onload = () => callback(script);
5   document.head.append(script);
6 }
7
8 function onLoadScript(script) {
9   if (!script) {
10     throw new Error('Ай-ай-ай!');
11   }
12 }
13
14 try {
15   loadScript('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js', onLoadScript);
16 } catch(err) {
17   // не попадем...
18 }
19
```

Я устала...

Future/Promise

Что такое Promise?



Promise

Promise (обещание) - это обертка, которая позволяет нам использовать переменные, значения которых нам неизвестны на момент создания обещания.

По-сути Promise позволяет асинхронный код организовывать так, будто он синхронный.

Далее подробнее

Более фундаментально

В информатике конструкции `future`, `promise` и `delay` в некоторых языках программирования формируют стратегию вычисления, применяемую для параллельных вычислений. С их помощью описывается объект, к которому можно обратиться за результатом, вычисление которого может быть не завершено на данный момент.

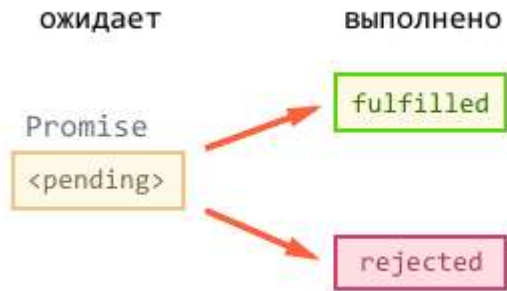
Где используется?

- Java (`java.util.concurrent.Future`)
- C++ (`std::future`)
- C# (`System.Threading.Tasks`)
- ...
- **JavaScript (Promise)**

Promise в JavaScript

Promises (промисы) — это специальные объекты, которые могут находиться в одном из трёх состояний:

- вначале **pending** («ожидание»)
- затем либо **fulfilled** («выполнено успешно»)
- либо **rejected** («выполнено с ошибкой»)



Как это выглядит в JavaScript?

```
1 const promise = new Promise(function(resolve, reject) {  
2     // Здесь можно выполнять любые действия  
3  
4     // вызов resolve(result) переведёт промис в состояние fulfilled  
5     // вызов reject(error) переведёт промис в состояние rejected  
6 });  
7  
8 // Можно создать сразу "готовый" промис  
9 const fulfilled = Promise.resolve(result);  
10 // const fulfilled = new Promise((resolve, _) => resolve(result));  
11 const rejected = Promise.reject(error);  
12 // const rejected = new Promise( (_, reject) => reject(error));  
13
```

Как это выглядит в JavaScript?

Основной способ взаимодействия с промисом это регистрация коллбеков для получения конечного результата промиса или сообщения о причине, по которой он не был выполнен. По-простому, на промисы можно навесить два коллбека:

- `onFulfilled` — срабатывают, когда `promise` находится в состоянии «выполнен успешно»
- `onRejected` — срабатывают, когда `promise` находится в состоянии «выполнен с ошибкой»

```
1 const promise = new Promise( ... );  
2  
3 // Можно навесить их одновременно  
4 promise.then(onFulfilled, onRejected);  
5  
6 // Можно по отдельности  
7 // Только обработчик onFulfilled  
8 promise.then(onFulfilled);  
9 // Только обработчик onRejected  
10 promise.then(null, onRejected);  
11 promise.catch(onRejected); // Или так  
12
```

```
1 const promise = new Promise(function(resolve, reject) {  
2     // do smth  
3     resolve('success'); // or  
4     // reject(new Error('failure'));  
5 });  
6  
7 promise  
8     .then(res => console.log(res))  
9     .catch(err => console.error(err));  
10
```


В чем профит?

- Можно навешивать несколько обработчиков-колбэков подряд
- Можно навесить обработчик-колбэк потом
- Можно передавать промисы в качестве аргументов в другие части системы
- Можно строить цепочки асинхронных вызовов без callback hell

Несколько независимых обработчиков

```
1 // 'cb1 success', 'cb2 success'
2 const promise = Promise.resolve('success');
3
4 promise.then(res => { console.log('cb1', res); }); // 1
5 promise.then(res => { console.log('cb2', res); }); // 2
6
```

Чейнинг

```
1 // 'value 1', 'value 2', 'value 3'
2 const promise = Promise.resolve('value 1');
3
4 const p2 = promise
5     .then(res => { console.log(res); return 'value 2'; }) // 1
6     .then(res => { console.log(res); return 'value 3'; }) // 2
7     .then(res => { console.log(res); });                  // 3
8
9 p2 === promise // false
10
```

Обработка асинхронных ошибок

```
1 // 'value 1', 'Error!', 'Error caught!'
2 const promise = Promise.resolve('value 1');
3
4 promise
5   .then(res => { console.log(res); throw 'Error!'; })           // 1
6   .then(res => { console.log('foo'); })
7   .then(res => { console.log('bar'); })
8   .then(res => { console.log('baz'); })
9   .catch(err => { console.error(err); return 'Error caught!'; }) // 2
10  .then(res => { console.log(res); });                             // 3
11
```

Из промиса можно возвращать промис!

```
1 // 'foo', 'baz', 'bar', 'foobar'
2 const promise1 = Promise.resolve('foo')
3   .then(res => { console.log(res); return 'bar'; });    // foo
4
5 const promise2 = Promise.resolve('baz')
6   .then(res => { console.log(res); return promise1; })  // baz
7   .then(res => { console.log(res); return 'foobar'; }) // bar
8   .then(res => { console.log(res); });                  // foobar
9
```

Промисификация

```
1 function loadScript(src, callback) {
2   return new Promise((resolve) => {
3     const script = document.createElement('script');
4     script.src = src;
5
6     script.onload = () => resolve(callback(script));
7     script.onerror = () => reject(new Error('Ай-яй-яй!'));
8
9     document.head.append(script);
10  });
11 }
12
13 loadScript('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js')
14   .then((script) => {alert(`Здорово, скрипт ${script.src} загрузился`)}))
15   .catch((error) => {alert(`Плохо дело, ошибка ${error}`)}));
16
```

Callback Hell?

```
1 loadScript('/my/script.js', function(script) {  
2  
3   loadScript('/my/script2.js', function(script) {  
4  
5     loadScript('/my/script3.js', function(script) {  
6       // ...и так далее, пока все скрипты не будут загружены  
7     });  
8  
9   })  
10  
11 }):
```

Лепота!

```
1 loadScript( '/my/script.js' )  
2   .then( ( ) => loadScript( '/my/script2.js' ) )  
3   .then( ( ) => loadScript( '/my/script3.js' ) )  
4   .then( ( ) => {} );  
5
```


Какие еще приколы с Promise есть?

- `Promise.all`
- `Promise.race`
- `Promise.any`
- `Promise.allSettled`

Promise.all

```
1 // Делаем что-нибудь асинхронное и важное параллельно
2 Promise.all([
3     PromiseGet('/user/1'),
4     PromiseGet('/user/2'),
5 ]).then(function(users) {
6     // Результатом станет массив из значений всех промисов
7     users.forEach(function(user, i) {
8         console.log(`User #${i}: ${value}`);
9     });
10 });
11
```

Promise.race

```
1 // Делаем что-нибудь асинхронное и важное наперегонки!  
2 Promise.race([  
3     promiseSomething(),  
4     promiseSomethingElse()  
5 ]).then(function(result) {  
6     // Результатом станет значение самого "быстрого" промиса  
7     console.log(`Result: ${value}`);  
8 });  
9
```

Promise.any

```
1 // вернет первое fulfilled, либо будет rejected с массивом причин
2 const promises = [
3   Promise.reject('ERROR A'),
4   Promise.reject('ERROR B'),
5   Promise.resolve('result'),
6 ]
7 Promise.any(promises)
8   .then((result) => console.log(result));
9   // result
10
```

Promise.allSettled

```
1 const promise1 = Promise.resolve(3);
2 const promise2 = new Promise(
3   (resolve, reject) => setTimeout(reject, 100, 'foo')
4 );
5 const promises = [promise1, promise2];
6 Promise.allSettled(promises).
7   then((results) => results.forEach((result) => console.log(result)));
8
9 // Вывод:
10 // { status: "fulfilled", value: 3 }
11 // { status: "rejected", reason: "foo" }
12
```

Добавим еще сахара

Async

```
1 async function f() {  
2     return 1;  
3 }  
4 // async-функции всегда возвращают promise  
5
```

Async

```
1 async function f1() {  
2     return 1;  
3 }  
4 async function f2() {  
5     return Promise.resolve(1);  
6 }  
7 f1().then(console.log) // 1  
8 f2().then(console.log) // 1  
9
```


Async/Await

```
1 async function f() {  
2   let p = new Promise((resolve)=> setTimeout(()=>resolve('done'), 1000))  
3   let result = await p; // будет ждать 1сек  
4   console.log(result)  
5 }  
6 // await нельзя использовать в обычных функциях  
7
```

Async/Await обработка ошибок

```
1 async function throwable() {
2   await Promise.reject(new Error('Oops')); // throw new Error('Oops');
3 }
4
5 async function f() {
6   try {
7     let response = await throwable();
8   } catch (error) {
9     console.log(error); // Oops
10  }
11 }
12
```

Fetch API

Fetch API

Метод `fetch` — это XMLHttpRequest нового поколения. Он предоставляет улучшенный интерфейс для осуществления запросов к серверу: как по части возможностей и контроля над происходящим, так и по синтаксису, так как построен на промисах

```
1 // Синтаксис метода fetch:  
2 const fetchPromise = fetch(url[, options]);  
3
```

Fetch API options

- `method` — метод запроса
- `headers` — заголовки запроса (объект)
- `body` — тело запроса: `FormData`, `Blob`, строка и т.п.
- `mode` — одно из: «`same-origin`», «`no-cors`», «`cors`», указывает, в каком режиме кросс-доменности предполагается делать запрос
- `credentials` — одно из: «`omit`», «`same-origin`», «`include`», указывает, пересылать ли куки и заголовки авторизации вместе с запросом
- `cache` — одно из «`default`», «`no-store`», «`reload`», «`no-cache`», «`force-cache`», «`only-if-cached`», указывает, как кешировать запрос

Fetch API

```
1 fetch('/courses', {  
2     method: 'POST',  
3     mode: 'cors',  
4     credentials: 'include',  
5     body: JSON.stringify({  
6         title: 'ПСП',  
7         authors: ['Толпаров Натан', 'Алехин Сергей']  
8     })  
9 });  
10
```

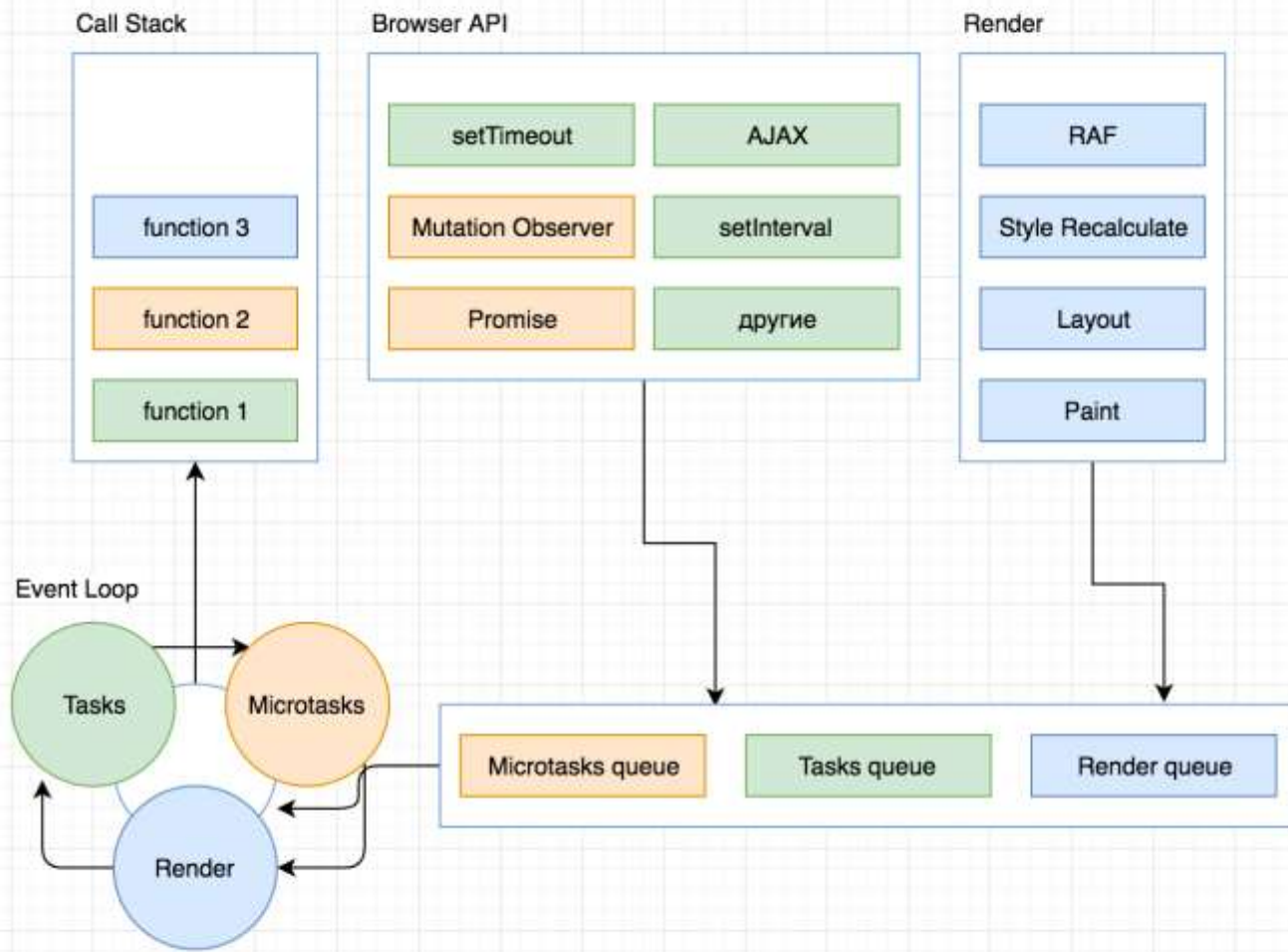
Тот самый дядя Event Loop

Что произойдет?

```
1 function foo() {  
2   setTimeout(foo, 0);  
3 }  
4  
5 foo();  
6
```


Что произойдет?

```
1 function foo() {  
2   Promise.resolve().then(foo);  
3 }  
4  
5 foo();  
6
```



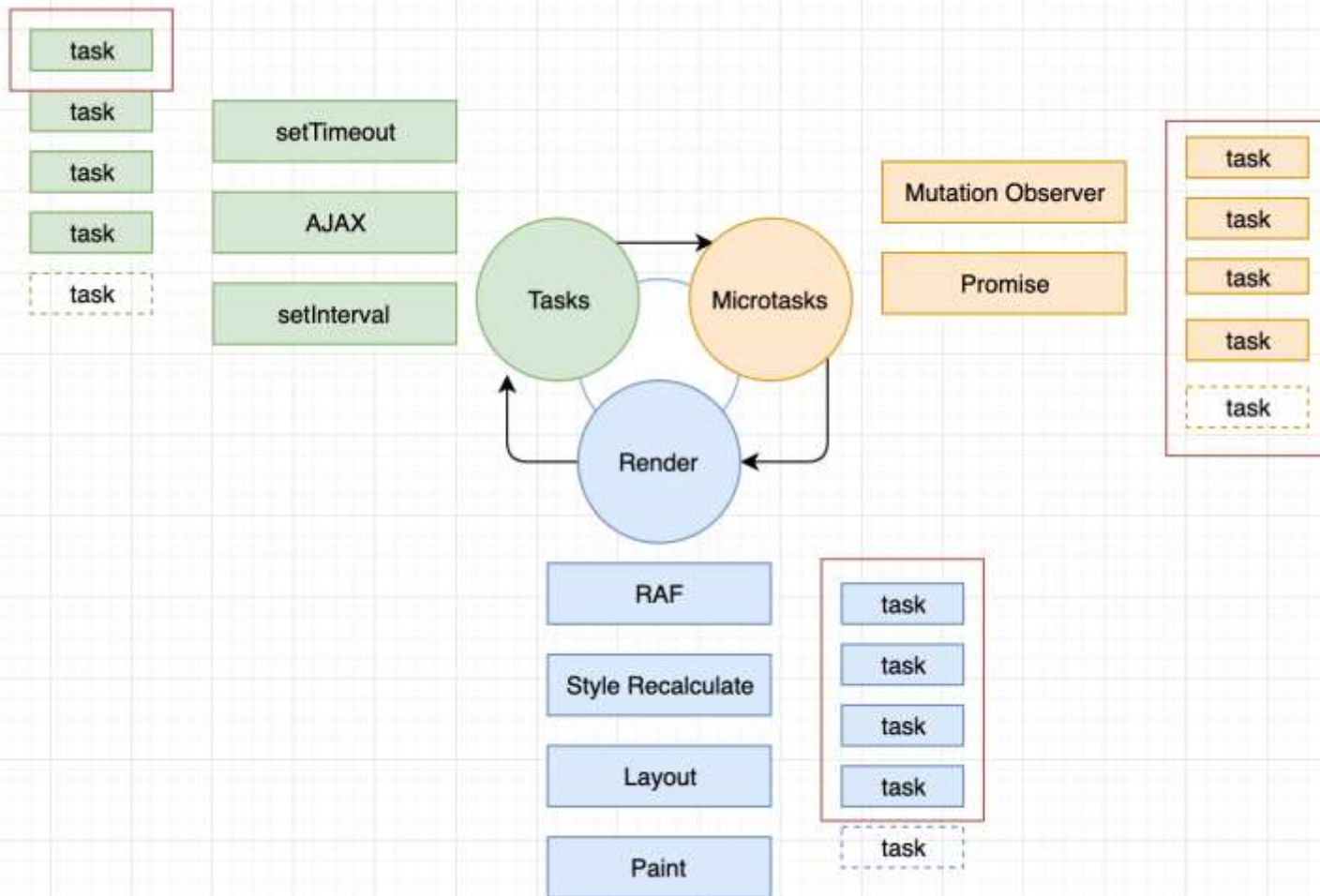
Как выполняются задачи?

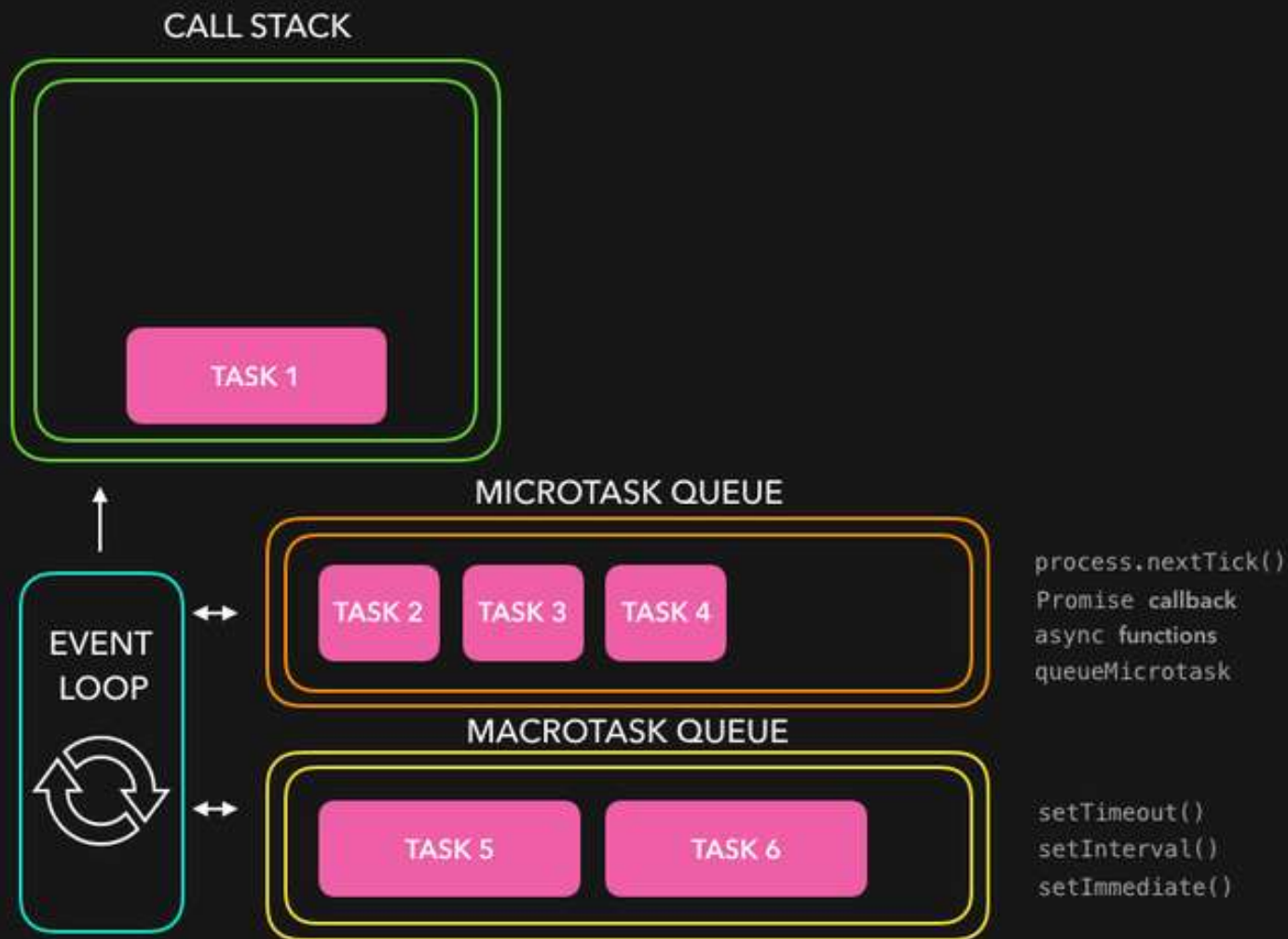
Мы видим, что единственное место, через которое задачи могут попасть в Call Stack и выполниться — это Event Loop. Т.е. Event Loop - это такой менеджер, который определяет какой задаче когда выполняться.

Какие виды задач есть?

Виды задач

- макротаски - любые функции отложенного вызова (callback);
- микротаски - промисы и MutationObserver
- рендеринг - Paint, Layout (из прошлой лекции), requestAnimationFrame, и тд.





Понятно?

Тогда решаем тупую задачу с собесов

```
1 (function() {  
2  
3   console.log('this is the start');  
4  
5   setTimeout(function cb() {  
6     console.log('setTimeout1');  
7   });  
8  
9   console.log('this is just a message');  
10  
11  Promise.resolve().then(() => {  
12    console.log('Promise.resolve1');  
13  })  
14  
15  setTimeout(function cb1() {  
16    console.log('setTimeout2');  
17  }, 0);  
18  
19  Promise.resolve()  
20    .then(() => {  
21      console.log('Promise.resolve2');  
22    }).then(() => {  
23      console.log('Promise.resolve3');  
24    })  
25  
26  console.log('this is the end');  
27 })();
```



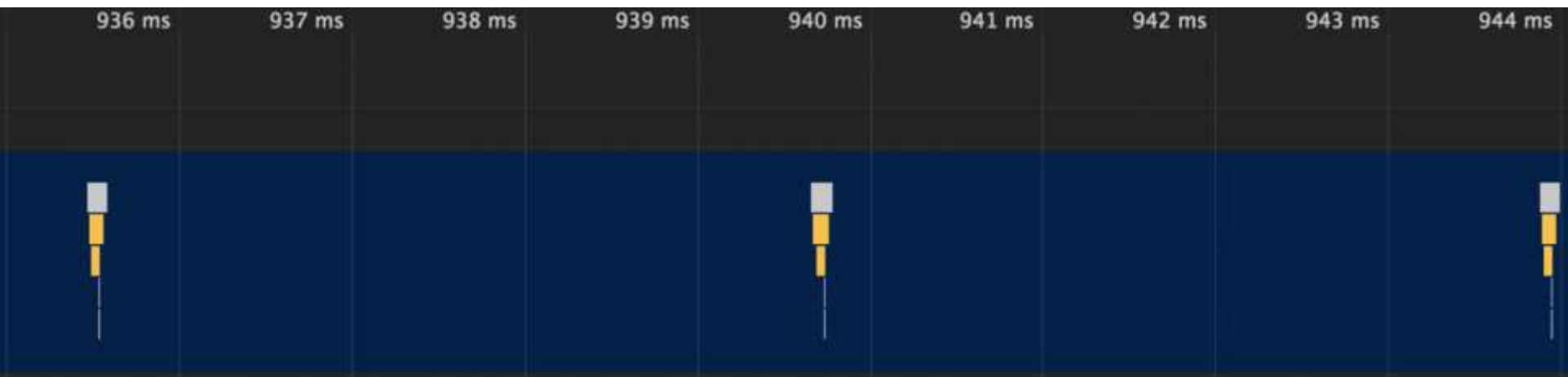
```
1 this is the start
2 this is just a message
3 this is the end
4 Promise.resolve1
5 Promise.resolve2
6 Promise.resolve3
7 setTimeout1
8 setTimeout2
9
```

Возвращаемся к изначальным вопросам

Что произойдет?

```
1 function foo() {  
2   setTimeout(foo, 0);  
3 }  
4  
5 foo();  
6
```

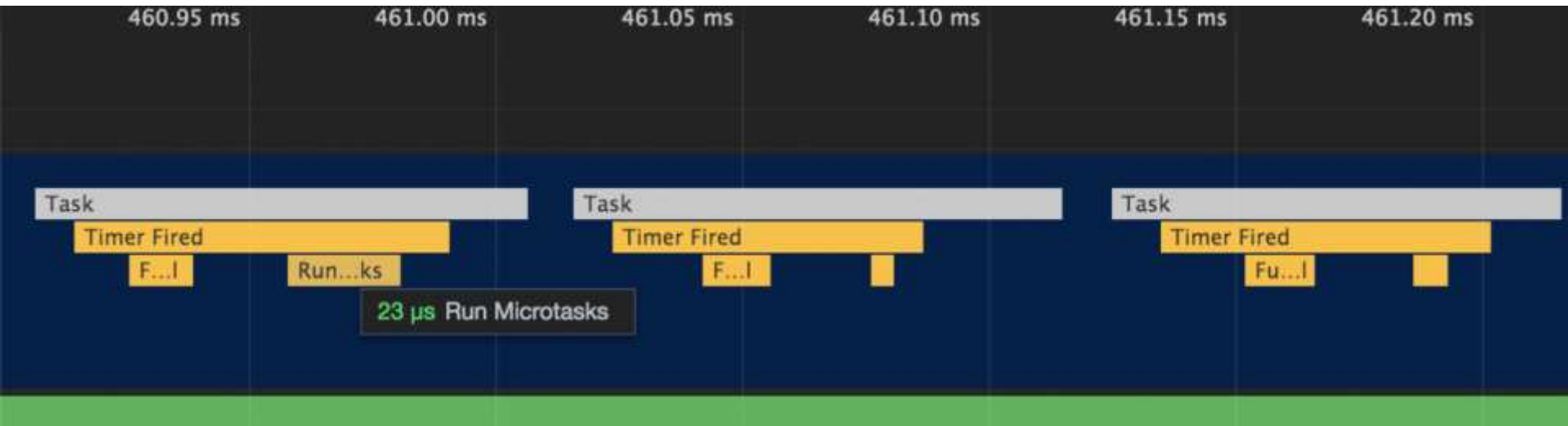
Макротаска в 4мс



Что произойдет?

```
1 function foo() {  
2   Promise.resolve().then(foo);  
3 }  
4  
5 foo();  
6
```

Микротаски друг за другом



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)

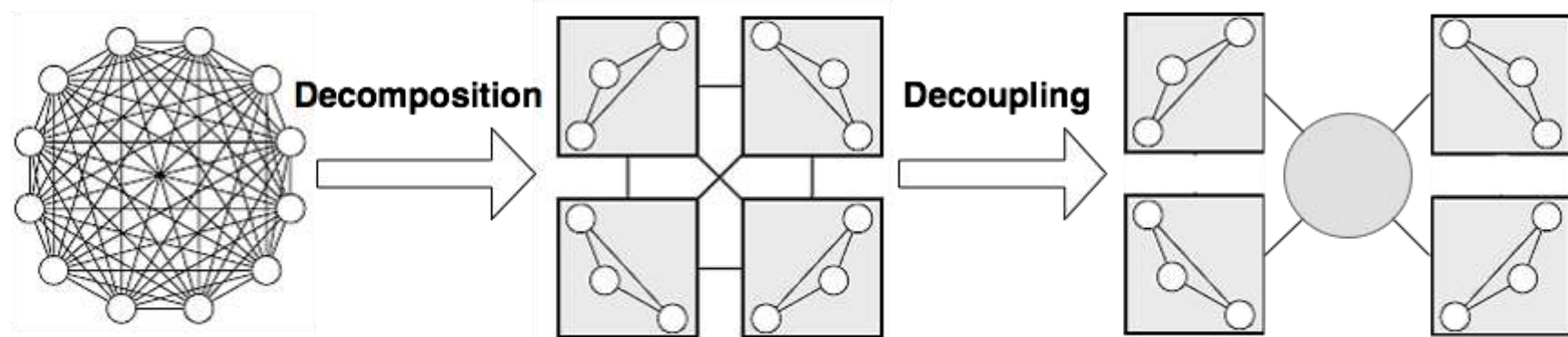
Методологии и принципы



Для создания хорошей
архитектуры используем
проверенные подходы

Декомпозиция

Создание Архитектуры

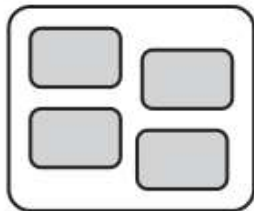


Хорошая декомпозиция -
иерархическая

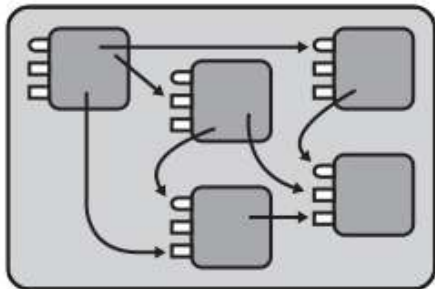
Декомпозиция (иерархическая)



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 сопряжены внутри (**high internal cohesion**) и минимально связаны друг с другом (**low external coupling**)

Модули, на которые разбивается система, должны быть, по возможности, независимы или слабо связаны 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

```
1 const service = new SomeService();
2 // функция-обработчик события
3 const onload = function (data) { console.log(data); }
4
5 // подписываемся на событие
6 service.on('loaded', onload);
7 service.emit('loaded', {data: 42});    // событие 1
8 service.emit('loaded', {foo: 'bar'});  // событие 2
9 // отписываемся от события
10 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)

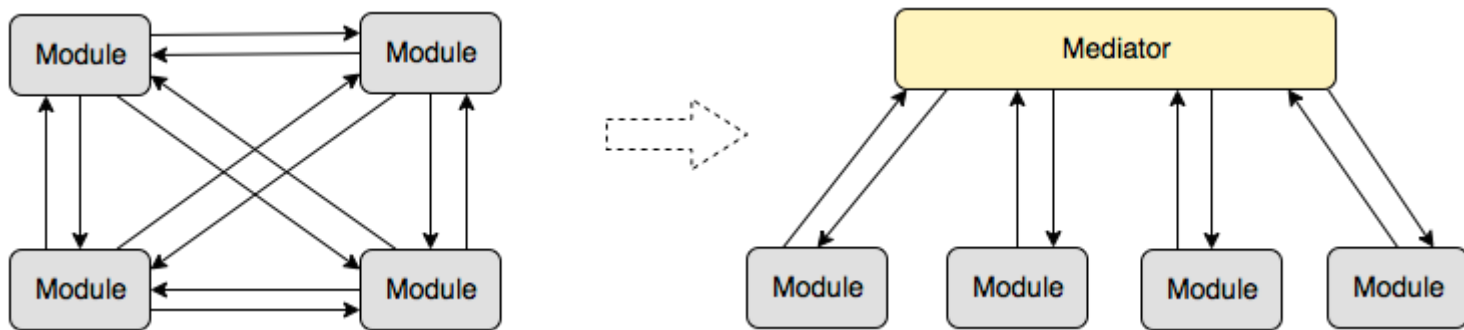
```
1 class ЧБПринтер {  
2     print (doc) { ... }  
3 }  
4  
5 class ЦветнойПринтер {  
6     print (doc) { ... }  
7 }  
8  
9 class Сканер {  
10     scan (doc) { ... }  
11 }
```

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)

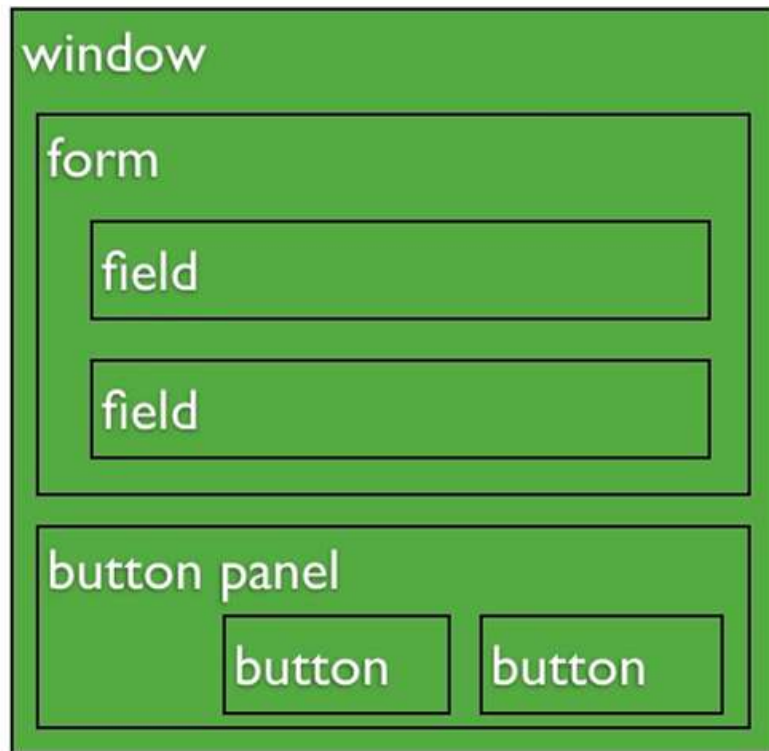
- Минимизировать использование глобальных зависимостей
- Не использовать неявные зависимости — все зависимости модуля должны быть обозначены явно
- Минимизировать связи между модулями — например, с помощью паттернов Observable и Mediator

Хорошая декомпозиция -
компонентная

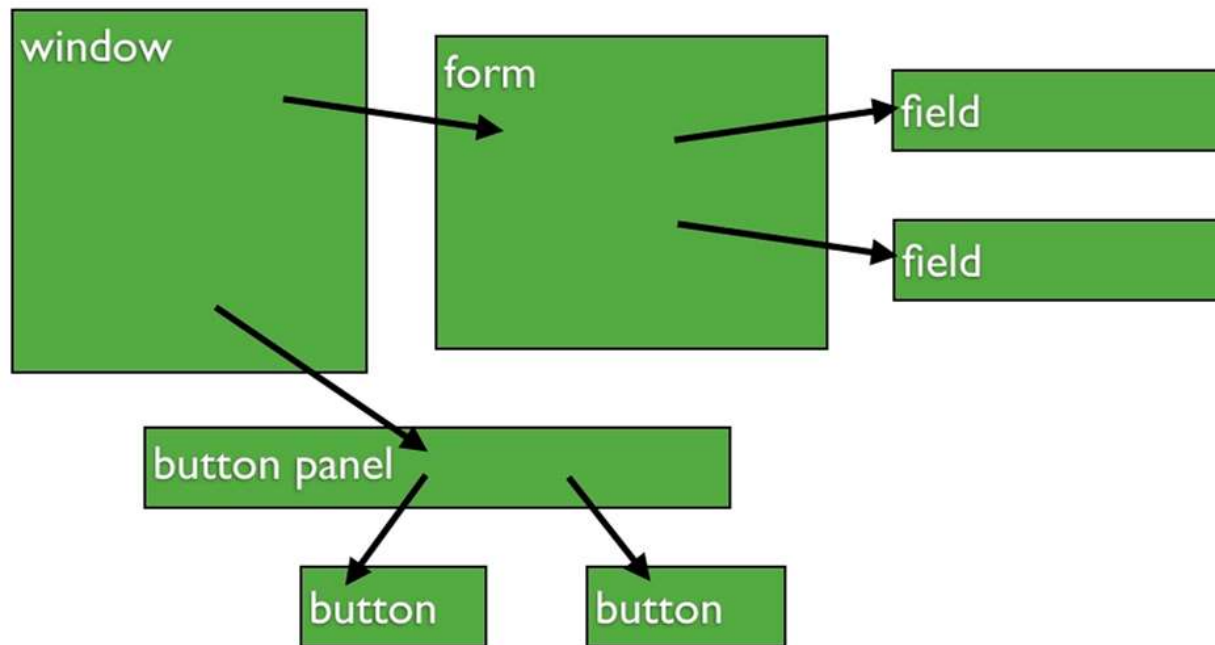
Декомпозиция (компоненты)

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

Обычный подход



Компонентный подход

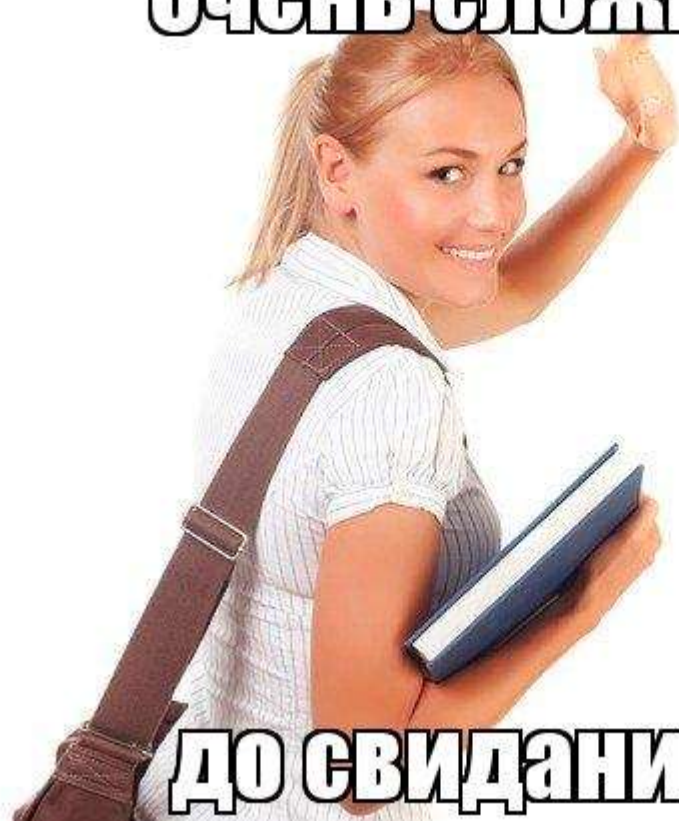


Декомпозиция (советы)

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

Очень сложно,



до свидания!

Архитектура JS

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

Как нам правильно
организовать код?

Шаблоны MVC

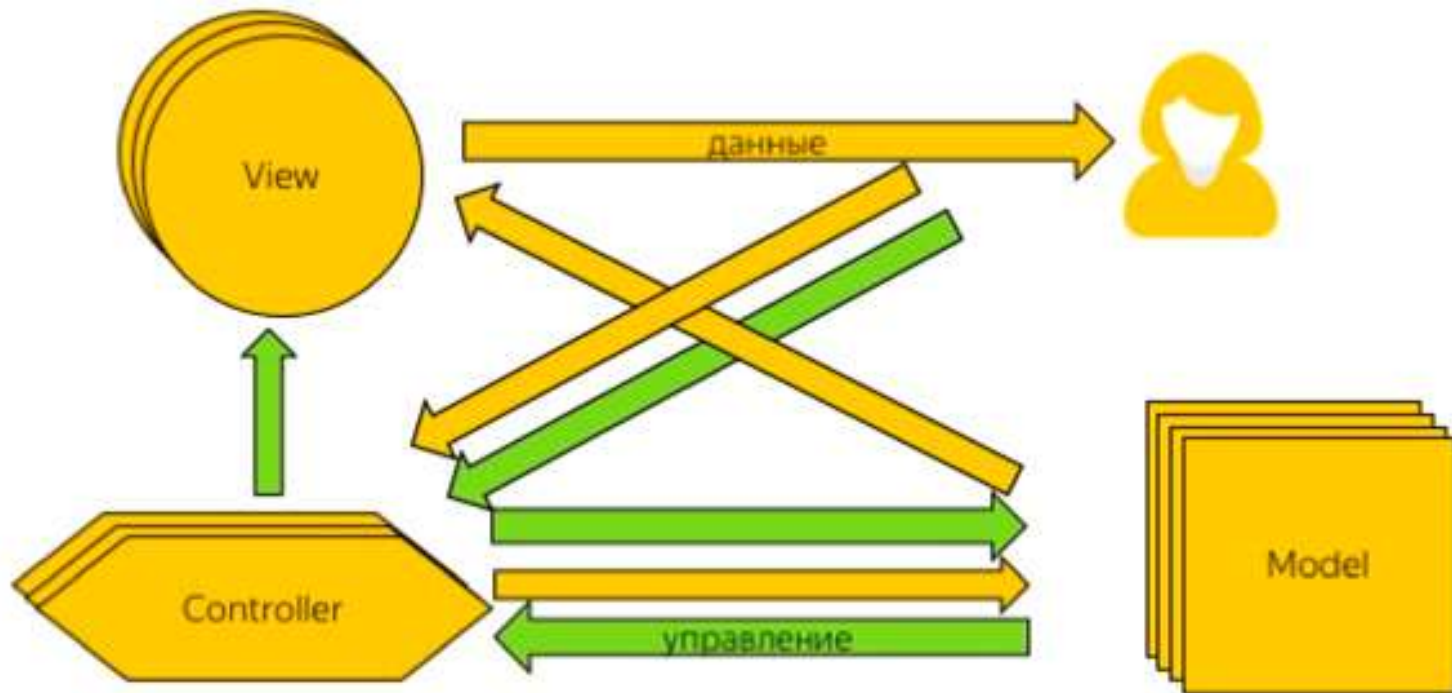


MV?

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 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

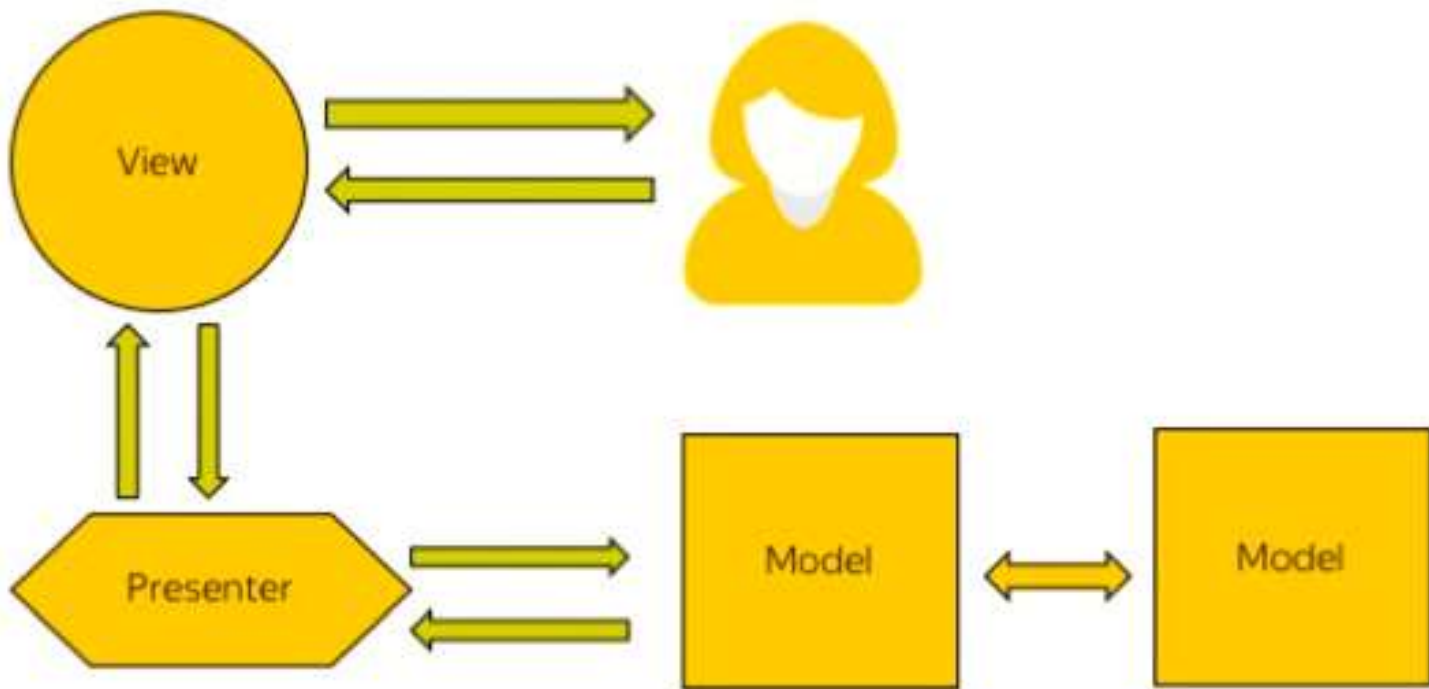
Пример: заполнение формы и отправка на сервер

- Controller и Model находятся на бэкенде, серверная шаблонизация View

Проблемы:

- Не подходит для SPA и активного ajax

Шаблоны MVP



Итоги MVP

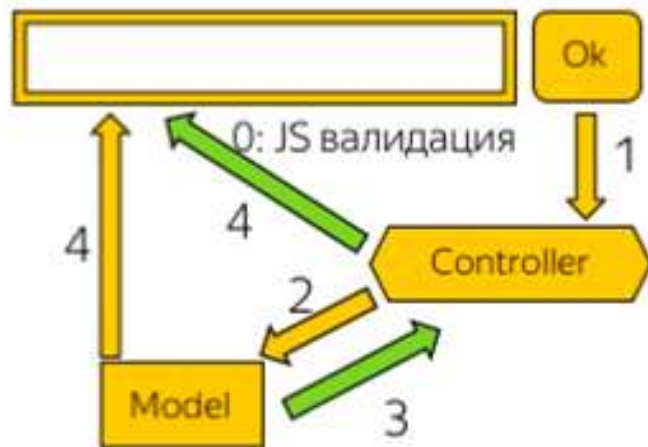
Пример: заполнение формы и отправка на сервер

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

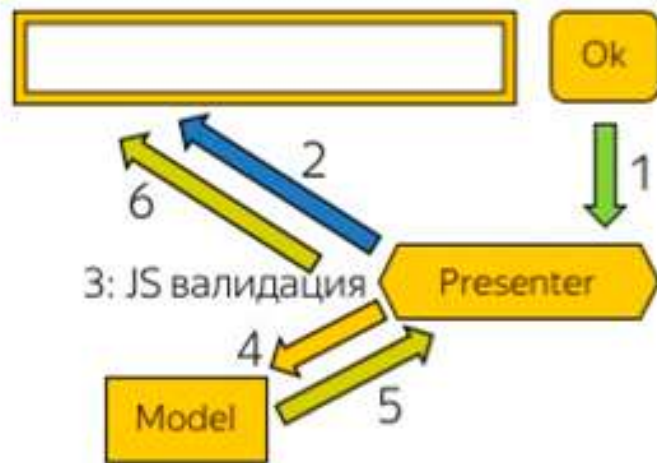
Проблемы:

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

Пример MVC/MVP



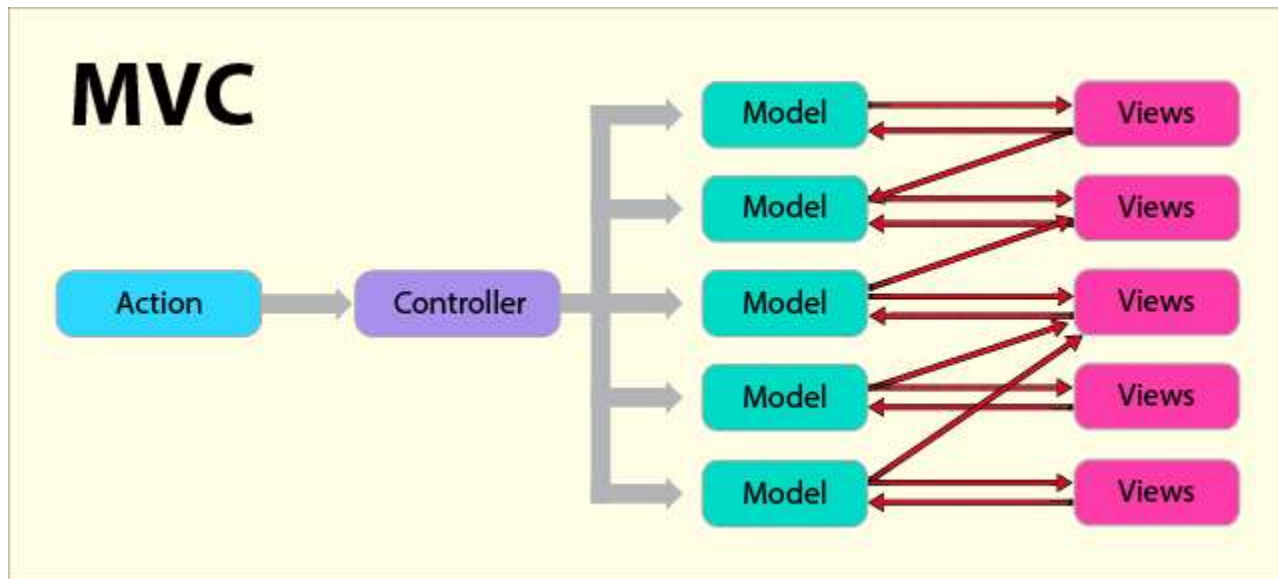
MVC



MVP

MVP решает
все проблемы ?

Hea



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.

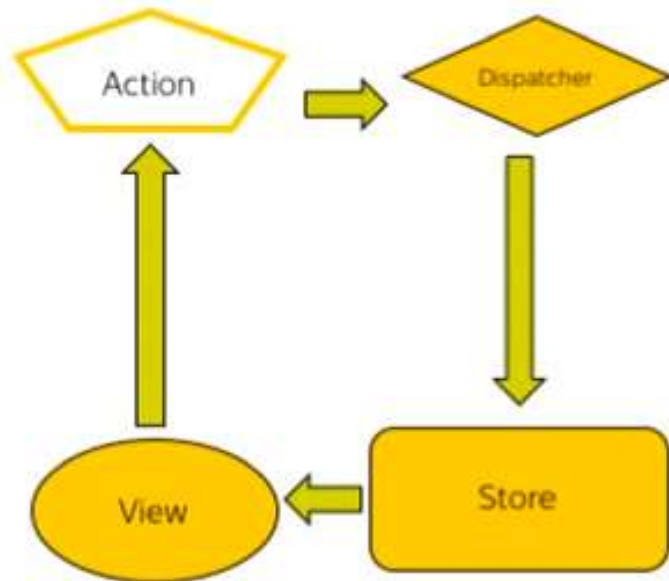
Основной отличительной особенностью Flux является односторонняя направленность передачи данных между компонентами Flux-архитектуры. Архитектура накладывает ограничения на поток данных, в частности, исключая возможность обновления состояния компонентов самими собой. Такой подход делает поток данных предсказуемым 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*



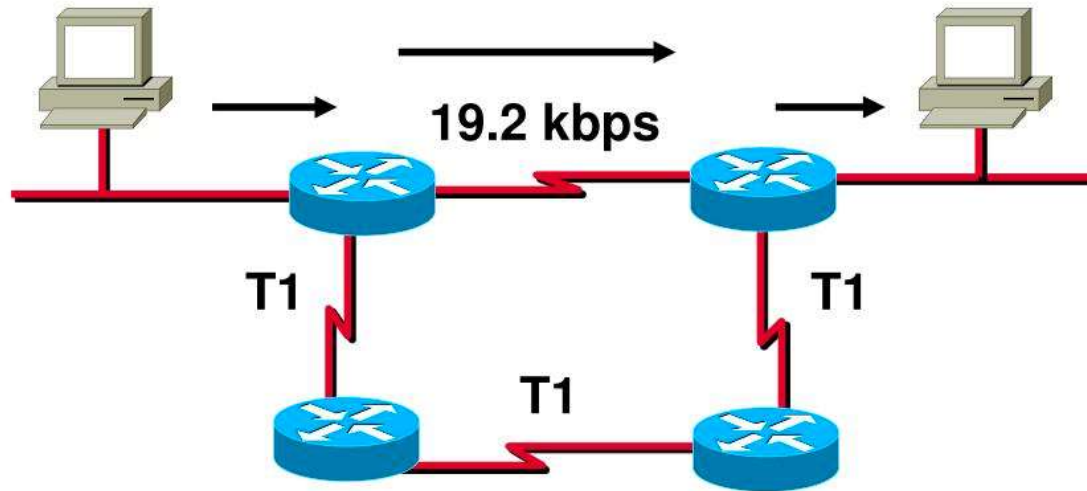
Пример Flux



Роутинг

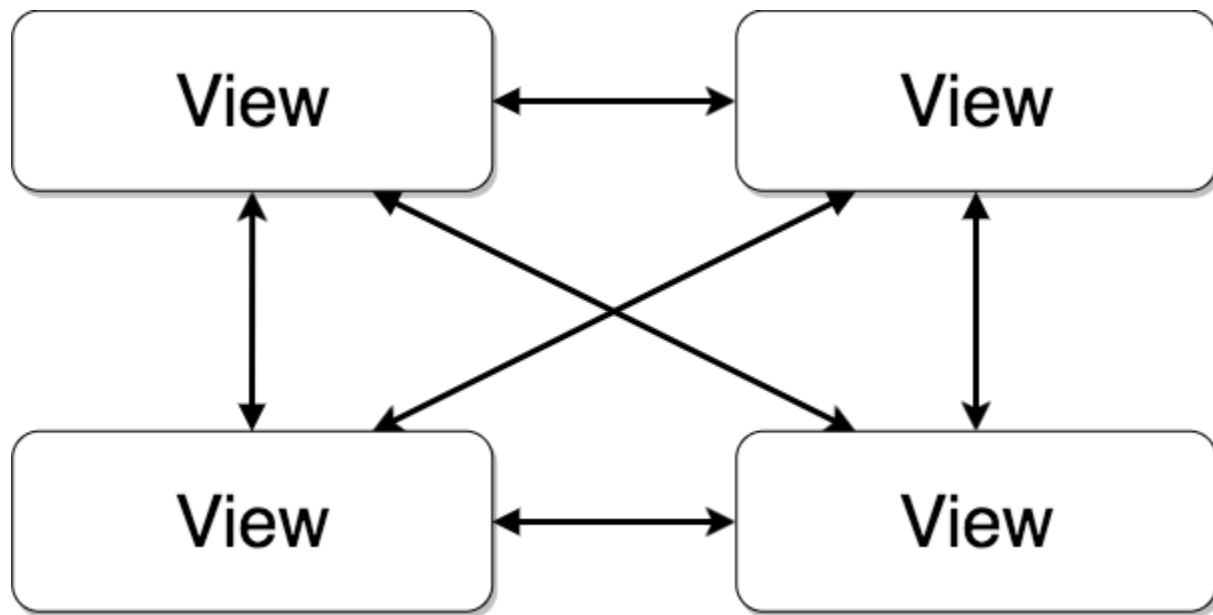
Подробнее тут <https://frontend.tech-mail.ru/slides/s5/>

Routing Protocol (RIP)



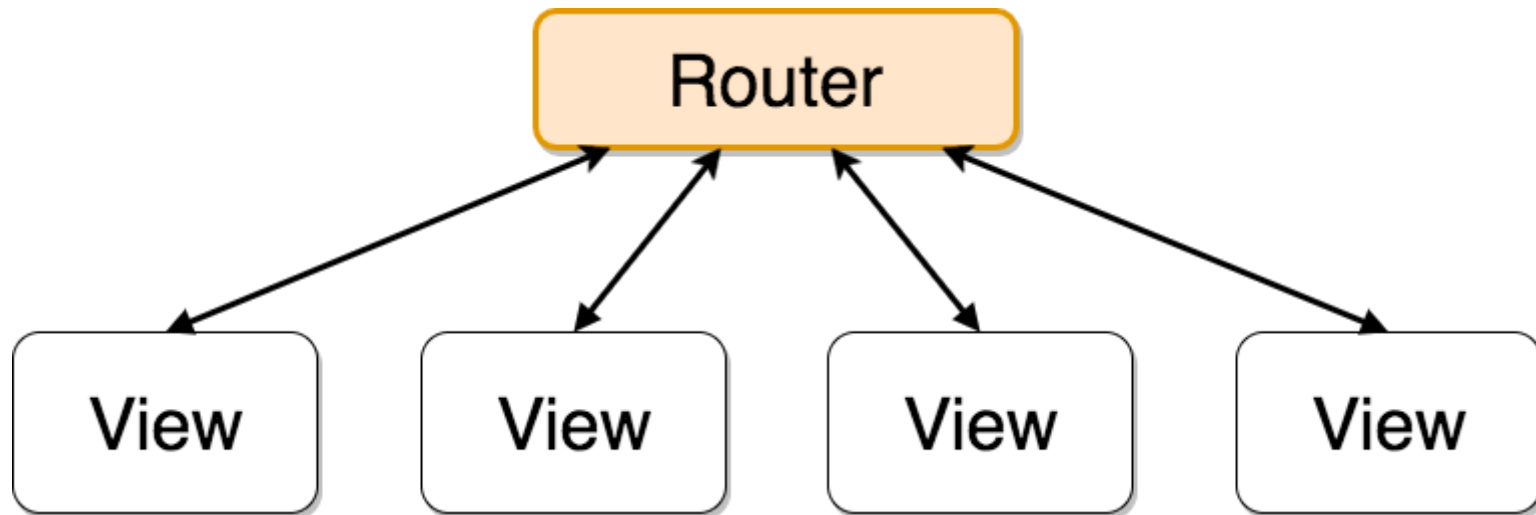
- Hop count metric selects the path
- Routes update every 30 seconds

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/>

CSS

IS

AWESOME

Как стили попадают на страницу

- Браузерные стили
- `link rel="stylesheet"`
- `тег style`
- атрибут `style`

Люблю CSS

```
1 /* Селекторы! */
2
3 *                               /* универсальный селектор */
4 div, span, a                  /* селекторы по имени тегов */
5 .class                        /* селекторы по имени классов */
6 #id                           /* селекторы по идентификаторам */
7 [type="text"], [src*="/img/"] /* селекторы по атрибутам */
8 :first-child, :visited, :nth-of-type(An+B), :empty ...
9 ::before, ::placeholder, ::selection, ::first-letter ...
10 a > a, a + a , a ~ a         /* вложенность и каскадирование */
11
```

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

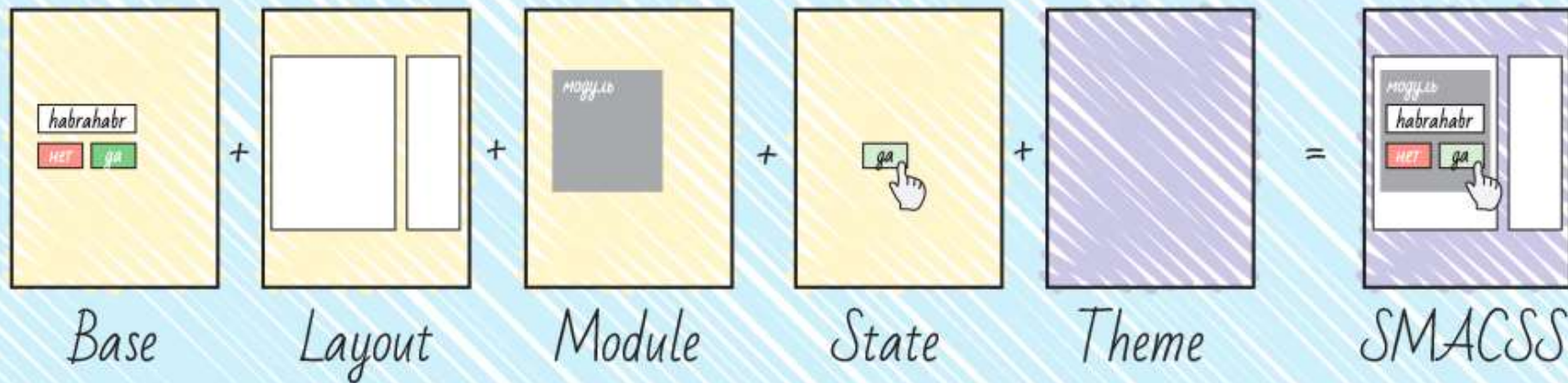
Принцип: внешний вид элемента не зависит от того, где он расположен. Вместо **.my-element button** создаем отдельный стиль **.control** для конкретного случая

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)

```
1 /* button.css */
2 .button {
3     width: 200px;
4     height: 48px;
5     border-radius: 12px;
6 }
7
8 .primary {
9     background-color: green;
10    font-weight: 500;
11 }
12
```

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
```

Хранение данных

Подробнее тут <https://learn.javascript.ru/data-storage>

Как хранить данные?

- Кеш браузера - сохраняет только ответы на запросы
- Cookies - подходит для хранения сессий, имеет маленький размер
- Web Storage API - механизм хранения key/value значений
- WebSQL/IndexedDB - база данных в браузере

Web Storage API

Web Storage API — механизм для сохранения key/value значений с возможностью программного управления данными. Предоставляет два host объекта в браузере пользователя с **возможностью персистентного хранения данных (до 10 MB на origin)**

- `window.sessionStorage` - сохраняет данные пока открыт браузер
- `window.localStorage` - сохраняет данные навсегда

Web Storage API (методы)

- `setItem(key, value)` - сохранить пару ключ/значение
- `getItem(key)` - получить данные по ключу `key`
- `removeItem(key)` - удалить данные с ключом `key`
- `clear()` - удалить все
- `key(index)` - получить на заданной позиции
- `length` - количество элементов в хранилище

LocalStorage (пример)

```
1 localStorage.setItem( 'name', 'John' )  
2 alert( localStorage.getItem( 'name' ) ) // John  
3  
4 localStorage.removeItem( 'name' )  
5 alert( localStorage.getItem( 'name' ) ) // null  
6
```


LocalStorage (JSON)

```
1 localStorage.setItem('user', JSON.stringify({name: "John"}))
2 alert( JSON.parse(localStorage.getItem('user')).name ) // John
3
4 localStorage.removeItem('user')
5 alert( localStorage.getItem('name') ) // null
```

LocalStorage (JSON)

```
1 function setJSON(key, value) {  
2     localStorage.setItem(key, JSON.stringify(value));  
3 }  
4 function getJSON(key) {  
5     const value = localStorage.getItem(key);  
6     return value ? JSON.parse(value) : null;  
7 }
```

SessionStorage (пример)

```
1 sessionStorage.setItem( 'name', 'John' )  
2 alert( sessionStorage.getItem( 'name' ) ) // John  
3  
4 sessionStorage.removeItem( 'name' )  
5 alert( sessionStorage.getItem( 'name' ) ) // null  
6
```

Web Storage API (событие)

Когда обновляются данные в `localStorage` или `sessionStorage`, генерируются событие `storage` со следующими свойствами:

- `key` - ключ, который обновился
- `oldValue` - старое значение
- `newValue` - новое значение
- `url` - url страницы, где произошло обновление
- `storageArea` - объект `localStorage` или `sessionStorage`

Web Storage API (событие)

```
1 // обработчик добавляется на объект window
2 window.addEventListener('storage', function (e) {
3     /* e.key, e.newValue */
4     ...
5 });
```

IndexedDB

IndexedDB — низкоуровневое API для клиентского хранилища большого объема структурированных данных, включая файлы/blobs. Эти API используют индексы для обеспечения высоко-производительного поиска данных. Максимальный размер сохраняемых данных — 50 MB

- хранит практически любые значения по ключам, несколько типов ключей
- поддержка транзакций
- поддерживает запросы в диапазоне ключей и индексы

IndexedDB (пример)

```
1 // открываем базу данных Forum (доступно и в воркерах!)
2 const request = window.indexedDB.open('Forum', 3); // 3 - версия бд
3
4 // обработчик успешного открытия базы данных
5 request.onsuccess = function(event) {
6     const db = event.target.result;
7     const store = db.createObjectStore('users', { keyPath: 'userId' });
8
9     store.createIndex('age', 'age', { unique: false });
10    store.createIndex('email', 'email', { unique: true });
11
12    store.add({ age: 21, email: 'test@test.ru' });
13 };
14
```

WebSQL

WebSQL — **полноценная SQL база данных**, которая позволяет персистентно **хранить данные в браузере пользователя** и работать с ними посредством SQL-запросов. Максимальный размер сохраняемых данных — 5 MB

WebSQL (пример)

```
1 // создаём объект базы данных (доступно и в воркерах!)
2 const db = openDatabase('forum', 'v1.0.0', 'Forum', 100000);
3
4 // создаём транзакцию
5 db.transaction(function(tx) {
6     tx.executeSql(
7         'SELECT COUNT(*) FROM `forum`',
8         [],
9         function (result) { console.log(result) },
10        function (tx, error) { /* some error logic */ }
11    );
12 });
13
```