

ОБЗОР СОВРЕМЕННЫХ ВОЗМОЖНОСТЕЙ JAVASCRIPT

ОПЕРАТОР LET С БЛОЧНОЙ ОБЛАСТЬЮ ВИДИМОСТИ

```
function func() {  
  if (true) {  
    let tmp = 123;  
  }  
  console.log(tmp); // ReferenceError: tmp is not defined  
}
```

```
function func() {  
  if (true) {  
    var tmp = 123;  
  }  
  console.log(tmp); // 123  
}
```

ОПЕРАТОР LET

```
function func() {  
  let foo = 5;  
  if (...) {  
    let foo = 10; // shadows outer `foo`  
    console.log(foo); // 10  
  }  
  console.log(foo); // 5  
}
```

ОПЕРАТОР CONST

```
let foo = 'abc';  
foo = 'def';  
console.log(foo); // def
```

```
const foo2 = 'abc';  
foo2 = 'def'; // TypeError
```

ОПЕРАТОР FOR OF

Перебор элементов массива (или любого объекта, для которого определен Iterator)

```
let arr = [1,2,3];  
  
for (let a of arr) {  
  console.log(a);  
}
```

СТРЕЛОЧНАЯ ФУНКЦИЯ

```
f = v => v + 1;
```

// аналог в обычном варианте

```
var f = function (v) { return v + 1; }
```

Пример использования:

```
var arr = [1,2,3];  
arr.forEach(i=>console.log(i));
```

СТРЕЛОЧНАЯ ФУНКЦИЯ С НЕСКОЛЬКИМИ ПАРАМЕТРАМИ

```
f = (x,y) => x+y;  
f(1,2) === 3;
```

ФУНКЦИЯ СТРЕЛКИ С ТЕЛОМ ФУНКЦИИ

```
f = (x,y) => {  
  console.log(x,y);  
  return x+y;  
}
```

ФУНКЦИИ МАССИВОВ: MAP/FILTER/REDUCE

```
let arr = [1,2,3];
```

```
arr.find(x=>x>2);
```

```
arr.findIndex(x=>x>2);
```

```
let a = ["first", "second", "third"];
```

```
a.filter(x=>x.length>5);
```

```
a.map(x=>x.length); // [5,6,5]
```

```
a.map(x=>x.length).reduce((x,y)=>x+y); // 16
```

```
a.map(x=>x.length).reduce((x,y)=>x+y)/a.length; // avg length
```

```
a.forEach(x=>console.log(x.length)); // 5 6 5
```

ПРИМЕР ПРИМЕНЕНИЯ MAP/FILTER/REDUCE

// найти людей старше 30 и напечатать их имена

```
persons = [{name:"Ivan",age:25}, {name:"Mary",age:35},  
  {name:"Stas", age:33}];  
persons.filter(p=>p.age>30)  
  .map(p=>p.name)  
  .forEach(n=>console.log(n)); // Mary Stas
```

// напечатать средний возраст

```
persons.map(p=>p.age).reduce((a,b)=>a+b)/persons.length // 31
```

ИСПОЛЬЗОВАНИЕ КЛАССОВ

Конструкция `class` – удобный «синтаксический сахар» для задания конструктора вместе с прототипом:

```
class Shape {  
  constructor (id, x, y) {  
    this.id = id  
    this.move(x, y)  
  }  
  move (x, y) {  
    this.x = x  
    this.y = y  
  }  
}
```


НАСЛЕДОВАНИЕ

Также мы имеем возможность использовать наследование, однако надо понимать, что здесь просто происходит присваивание значения свойства prototype.

```
class Rectangle extends Shape {  
  constructor (id, x, y, width, height) {  
    super(id, x, y)  
    this.width = width  
    this.height = height  
  }  
}  
class Circle extends Shape {  
  constructor (id, x, y, radius) {  
    super(id, x, y)  
    this.radius = radius  
  }  
}
```

ДОСТУП К РОДИТЕЛЬСКОМУ КЛАССУ С ПОМОЩЬЮ SUPER

```
class Shape {  
    ...  
    toString () {  
        return `Shape(${this.id})`  
    }  
}  
class Rectangle extends Shape {  
    constructor (id, x, y, width, height) {  
        super(id, x, y)  
        ...  
    }  
    toString () {  
        return "Rectangle > " + super.toString()  
    }  
}  
class Circle extends Shape {  
    constructor (id, x, y, radius) {  
        super(id, x, y)  
        ...  
    }  
    toString () {  
        return "Circle > " + super.toString()  
    }  
}
```

СТАТИЧЕСКИЕ ЧЛЕНЫ КЛАССА

Мы можем создавать статические методы в классе – внутри них нельзя обращаться к `this`.

```
class Circle extends Shape {  
    static defaultCircle () {  
        return new Circle("default", 0, 0, 100)  
    }  
}  
  
var defRectangle = Rectangle.defaultRectangle()  
var defCircle    = Circle.defaultCircle()
```

В этом примере метод `defaultCircle` является, по сути, фабричным методом, создающим экземпляры `Circle`.

ГЕТТЕРЫ И СЕТТЕРЫ

```
class Rectangle {  
    constructor (width, height) {  
        this._width = width  
        this._height = height  
    }  
    set width (width) { this._width = width }  
    get width () { return this._width }  
    set height (height) { this._height = height }  
    get height () { return this._height }  
    get area () { return this._width * this._height }  
}  
var r = new Rectangle(50, 20)
```

```
r.area === 1000
```

Здесь вызван метод `get area()`, который вычислил значение.
При этом у нас нет возможности записать значение в `area`.

ОПЕРАТОР SPREAD ДЛЯ МАССИВА

Такой код позволит вставить значения 3 и 4 в середину массива:

```
var mid = [3, 4];  
var arr = [1, 2, ...mid, 5, 6];  
console.log(arr); // [1, 2, 3, 4, 5, 6]
```

Также мы можем использовать спред-оператор для копирования массива:

```
var arr = [3, 4];  
var arr2 = [...arr]; // мы получили копию массива, таким образом изменения arr2 не изменят arr  
arr2.push(5); // arr2: [3,4,5], arr: [3,4]
```

Использование spread для добавления массива дважды:

```
var arr1 = [0, 1, 2];  
var arr2 = [3, 4, 5];  
arr1 = [...arr2, ...arr1]; // arr1 теперь [3, 4, 5, 0, 1, 2]
```

ОПЕРАТОР SPREAD ДЛЯ ОБЪЕКТА

В ES2018 появилась возможность использовать оператор ... и для объектов. Таким образом, он становится более лаконичной заменой Object.assign():

```
let obj = {a:1, b:2};  
let objClone = {...obj};
```

Также мы можем добавлять и перезаписывать свойства:

```
let obj = {a:1, b:2};  
let obj2 = {...obj, {b:3, c:4} };  
console.log(obj2); // {a:1, b:3, c:4}
```

Еще один пример:

```
var obj1 = { foo: 'bar', x: 42 };  
var obj2 = { foo: 'baz', y: 13 };  
var clonedObj = { ...obj1 };  
// Object { foo: "bar", x: 42 }  
var mergedObj = { ...obj1, ...obj2 };  
// Object { foo: "baz", x: 42, y: 13 }
```

ДЕСТРУКТУРИЗАЦИЯ МАССИВА

Деструктуризация (destructuring assignment) – это особый синтаксис присваивания, при котором можно присвоить массив или объект сразу нескольким переменным, разбив его на части.

```
let [firstName, lastName] = ["Василий", "Пупкин"];
```

```
alert(firstName); // Василий  
alert(lastName); // Пупкин
```

При таком присвоении первое значение массива пойдёт в переменную `firstName`, второе – в `lastName`, а последующие (если есть) – будут отброшены.

Ненужные элементы массива также можно отбросить, поставив лишнюю запятую:

```
// первый и второй элементы не нужны  
let [, , surname] = "Василий Васильевич Пупкин".split(" ");  
  
alert(surname); // Пупкин
```

В коде выше первый и второй элементы массива никуда не записались, они были отброшены. Как, впрочем, и все элементы после третьего.

ОПЕРАТОР SPREAD ПРИ ДЕСТРУКТУРИЗАЦИИ

Если мы хотим получить и последующие значения массива, но не уверены в их числе – можно добавить ещё один параметр, который получит «всё остальное», при помощи оператора "..." («spread», троеточие):

```
let [firstName, lastName, ...rest] = "Василий Васильевич Пупкин Великий".split(" ");
```

```
alert(firstName); // Василий
```

```
alert(lastName); // Васильевич
```

```
alert(rest); // Пупкин, Великий (массив из 2х элементов)
```

Значением rest будет массив из оставшихся элементов массива. Вместо rest можно использовать и другое имя переменной, оператор здесь – троеточие. Оно должно стоять только последним элементом в списке слева.

ЗНАЧЕНИЯ ПО УМОЛЧАНИЮ ПРИ ДЕСТРУКТУРИЗАЦИИ

Если значений в массиве меньше, чем переменных – ошибки не будет, просто присвоится **undefined**:

```
let [firstName, lastName] = [];
```

```
alert(firstName); // undefined
```

В таких случаях задают значение по умолчанию. Для этого нужно после переменной использовать символ = со значением, например:

```
// значения по умолчанию
```

```
let [firstName="Гость", lastName="Анонимный"] = [];
```

```
alert(firstName); // Гость
```

```
alert(lastName); // Анонимный
```

ЗНАЧЕНИЯ ПО УМОЛЧАНИЮ ПРИ ДЕСТРУКТУРИЗАЦИИ

В качестве значений по умолчанию можно использовать не только примитивы, но и выражения, даже включающие в себя вызовы функций:

```
function defaultLastName() {  
  return Date.now() + '-visitor';  
}
```

```
// lastName получит значение, соответствующее текущей дате:  
let [firstName, lastName=defaultLastName()] = ["Вася"];
```

```
alert(firstName); // Вася  
alert(lastName); // 1436...-visitor
```

ДЕСТРУКТУРИЗАЦИЯ ОБЪЕКТА

Деструктуризацию можно использовать и с объектами. При этом мы указываем, какие свойства в какие переменные должны «идти».

Базовый синтаксис:

```
let {var1, var2} = {var1: ..., var2: ...};
```

Пример использования:

```
let options = {  
  title: "Меню",  
  width: 100,  
  height: 200  
};
```

```
let {title, width, height} = options;
```

```
alert(title); // Меню  
alert(width); // 100  
alert(height); // 200
```

Объект справа – уже существующий, который мы хотим разбить на переменные. А слева – список переменных, в которые нужно соответствующие свойства записать.



ИММУТАБЕЛЬНОСТЬ

ИММУТАБЕЛЬНОСТЬ

- ♦ Самым простым и быстрым способом проверить, изменился ли объект, оказывается проверка ссылки на объект — изменилась она или нет.
- ♦ В этом случае можно не делать детальное сравнение свойств, например такое:
 - `_.isEqual(object1, object2)`
 - Намного быстрее и проще в случае изменения объекта заменять его, а не редактировать. Тогда проверку можно максимально упростить:
 - `object1 === object2`
 - В этом и заключается основная идея «неизменяемости». Дело не в том, что невозможно изменить объект.
 - Можно просто следовать правилу неизменяемости: «Если нужно изменить какой-то объект, замените его».

ИММУТАБЕЛЬНОСТЬ МАССИВОВ

Неизменяемость массивов: `[...array]`, `concat`, `slice`, `splice`, `filter`, `map`

```
let arr = [1,2,3];
```

```
let arr2 = arr; // таким образом мы создаем копию ссылки, и при внесении изменений  
                // будет меняться исходный массив, например:
```

```
arr2.push(10); // в результате изменился и arr: arr==[1,2,3,10]
```

```
let arr3 = [...arr]; // создание копии массива
```

```
arr3.push(10); // теперь мы меняем только копию
```

```
console.log(arr); // [1,2,3]
```

```
console.log(arr3); // [1,2,3,10]
```

Для иммутабельного добавления элемента в массив мы можем просто использовать спред-оператор:

```
let arr4 = [...arr, 10]; // иммутабельно добавляем в конец массива элемент 10: arr не меняется
```

```
let arr5 = [10, ...arr]; // иммутабельно добавляем в начало массива элемент 10
```

ИММУТАБЕЛЬНОСТЬ ОБЪЕКТОВ

Обычные операции над объектами – например, присвоение значения свойству, меняют объект.

Для иммутабельного изменения мы можем использовать функцию `Object.assign`.

`Object.assign(target, src1, src2...)` – копирует все свойства из `src1`, `src2` и т.д. в объект `target`.

```
let user = { name: "Бася" };
```

```
let visitor = { isAdmin: false, visits: true };
```

```
let admin = { isAdmin: true };
```

```
Object.assign(user, visitor, admin);
```

```
// user <- visitor <- admin
```

```
console.log( JSON.stringify(user) ); // name: Бася, visits: true, isAdmin: true
```

ИММУТАБЕЛЬНОСТЬ ОБЪЕКТОВ

Но `Object.assign()` можно использовать для 1-уровневого (не глубокого!) клонирования объектов:

```
let user = { name: "Вася", isAdmin: false };  
  
// clone = пустой объект + все свойства user  
  
let clone = Object.assign({}, user);
```

Таким образом, мы получаем клон объекта `user`.

Мы можем использовать это для получения объекта, в котором изменено значение 1 или нескольких свойств. Например, нам нужно изменить значение `name`:

```
let user2 = Object.assign({}, user, {name: "Дима"});
```

Теперь `user2` содержит значение `{name: "Дима", isAdmin: false }`

При этом объект `user` остался неизменным.

Также мы можем изменить значения сразу 2 полей, например:

```
let user3 = Object.assign({}, user, {name: "Иван", isAdmin: true});
```

Теперь у нас получился новый объект `user3`, при этом `user` остался неизменным.

ИММУТАБЕЛЬНОСТЬ ОБЪЕКТОВ: СПРЕД ОПЕРАТОР

Также можно использовать спред оператор – новый синтаксис для создания копий (должен войти в EcmaScript 2018).

Тогда синтаксис создания копии объекта упрощается и становится похожим на синтаксис создания копии массива:

```
let user2 = {...user, name: "Дима"};
```

Теперь user2 содержит значение {name: "Дима", isAdmin: false }

При этом объект user остался неизменным.

```
let user3 = {...user, name: "Иван", isAdmin: true};
```

Теперь у нас получился новый объект user3, при этом user остался неизменным.

Даже после повсеместного введения нового стандарта, останется много унаследованного кода, который продолжит использование метода Object.assign(). Поэтому необходимо знать и уметь применять оба варианта.

ЧИСТАЯ ФУНКЦИЯ

- Что такое чистая функция? Функция считается чистой, если она соответствует следующим утверждениям:
- При одинаковых аргументах результат вычисления будет одним и тем же. Никаких сюрпризов. Никаких побочных эффектов. Никаких вызовов API. Никаких изменений. Только вычисление.
- Примеры:

```
var values = {a: 1};  
function impureFunction(items) {  
  var b = 1;  
  items.a = items.a * b + 2;  
  return items.a;  
}  
var c = impureFunction(values)
```

```
var values = {a: 1};  
function pureFunction(a) {  
  var b = 1;  
  a = a * b + 2;  
  return a;  
}  
var c = pureFunction(values.a)
```

params → **function** → return value

ПРЕИМУЩЕСТВА ИММУТАБЕЛЬНОСТИ

1. Простое и быстрое отслеживание изменений

Для того, чтобы понять, одинаковы ли объекты `obj1` и `obj2`, необходимо провести их глубокое сравнение, что является дорогостоящей операцией.

Если же все преобразования были иммутабельны, в этом нет необходимости: если ссылки на объекты не совпадают, то и объекты — разные.

ПРИМЕР ИСПОЛЬЗОВАНИЯ ИММУТАБЕЛЬНОСТИ

Пусть у нас есть функция, ответственная за демонстрацию изменений

```
function showChanges(oldValue, newValue) {  
  if (oldValue == newValue) return; // ничего не изменилось – можем ничего не делать  
  // такой подход лежит в основе Virtual DOM и дает возможность экономить время  
  
  // ... логика отрисовки данных с учетом изменений  
}
```

```
let oldValue = obj1; // сохраняем исходное значение  
obj1.a = 10; // не иммутабельное изменение
```

```
showChanges(oldValue, obj1); // хотя объект изменился, ссылка осталась  
                               // той же – в результате изменения не будут отображены
```

```
let obj1 = {...obj1, a:10}; // иммутабельное изменение obj1  
showChanges(oldValue, obj1); // теперь мы знаем, что obj1 изменился  
                               // и должен быть перерисован
```

ПРЕИМУЩЕСТВА ИММУТАБЕЛЬНОСТИ

2. Безопаснее использовать

Переданные в функцию данные могут быть случайно испорчены, и отследить такие ситуации очень сложно.

При использовании иммутабельности мы не боимся, что данные могут случайно измениться — если какая-то функция меняет данные, она создает их новую копию.

3. Легче тестировать

Так как чистые функции зависят только от входных параметров, мы можем протестировать различные комбинациями параметров, и быть уверенными, что функция не зависит от других частей системы.

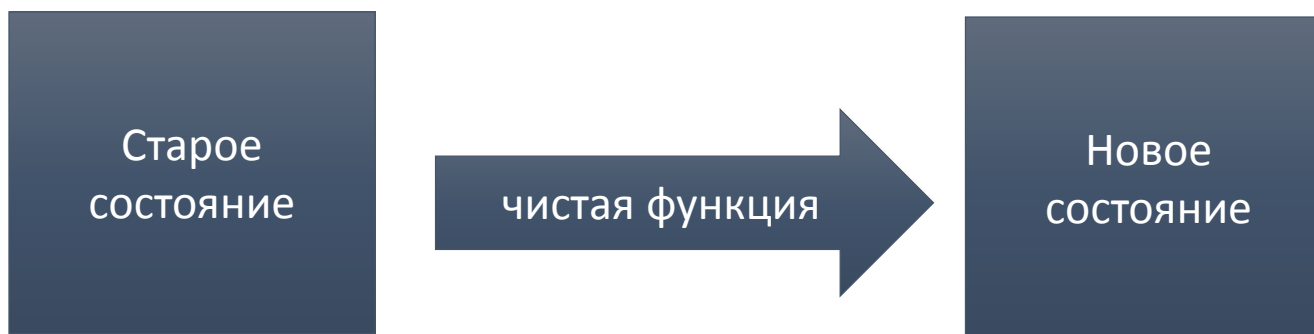
4. Легче мемоизировать

Мемоизация — сохранение результатов выполнения функций для предотвращения повторных вычислений. Это один из способов оптимизации, применяемый для увеличения скорости выполнения компьютерных программ. Перед вызовом функции проверяется, вызывалась ли функция ранее: если не вызывалась, функция вызывается и результат её выполнения сохраняется; если вызывалась, используется сохранённый результат.

ПРЕИМУЩЕСТВА ИММУТАБЕЛЬНОСТИ

5. Легче отлаживать

Отладка – очень существенный вопрос для больших систем. Когда данные иммутабельны, гораздо легче отслеживать изменения состояний системы:

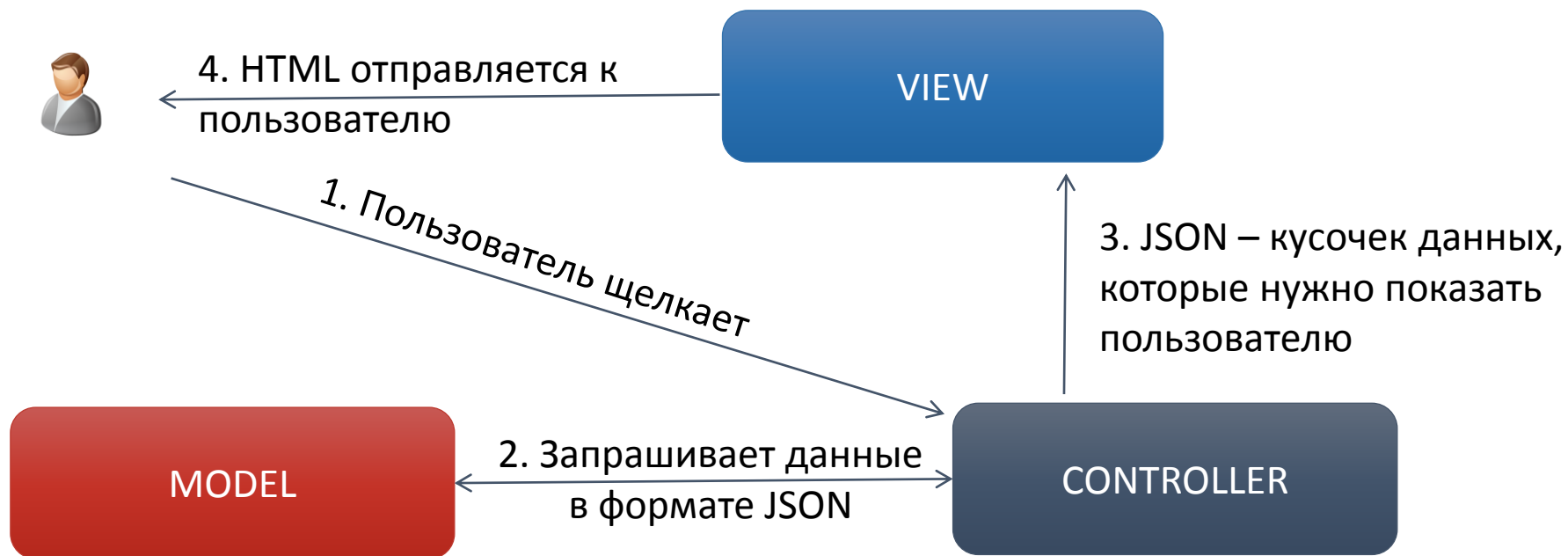


При этом мы можем видеть и отслеживать все изменения.

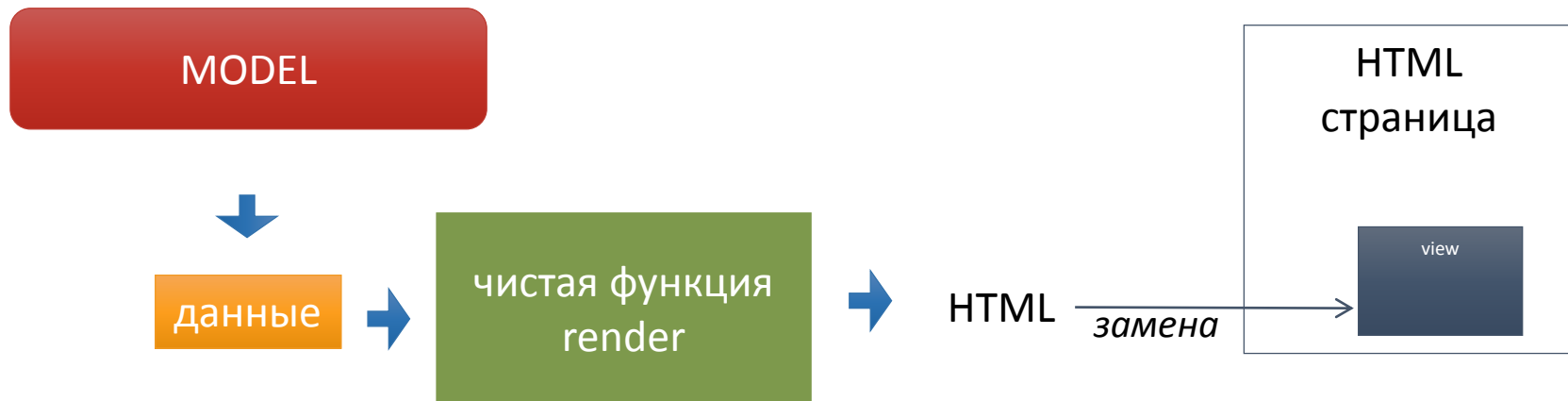
Таким образом, если происходит ошибка, ее гораздо проще отследить, потому что у нас есть вся полнота знаний о состоянии и преобразования, которые не содержат побочных эффектов.



МОДЕЛЬ MVC



ИММУТАБЕЛЬНЫЕ ФУНКЦИИ ДЛЯ VIEW



employee -> HTML

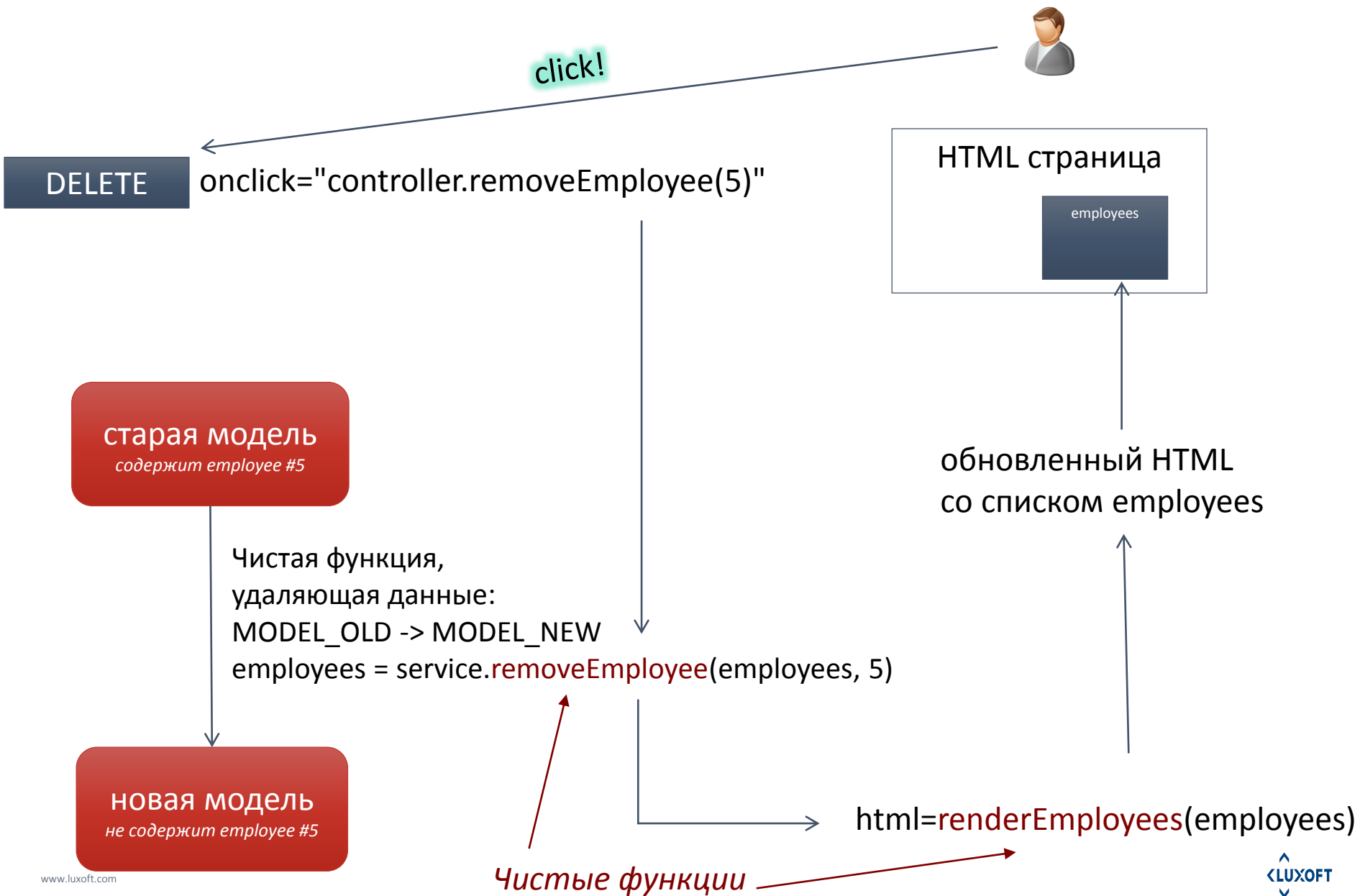
```
function renderEmployee(employee) {
  return `- ${employee.name} ${employee.surname}
    <button onclick="removeEmployee(${employee.id})">удалить</button>
  </li>`;
}

```

employees -> HTML

```
function renderEmployees(employees) {
  return "<ul>" + employees.map(e=>renderEmployee(e)).join("") + "</ul>";
}
```

ПРИМЕР: УДАЛЕНИЕ ИЗ СПИСКА СОТРУДНИКОВ





АСИНХРОННОСТЬ И РАБОТА С СЕРВЕРОМ

СИНХРОННЫЙ И АСИНХРОННЫЙ КОД

синхронный код

Команды синхронного кода выполняются в том порядке, в котором они следуют в тексте программы:

```
console.log('1');  
console.log('2');  
console.log('3');
```

Результат:

1
2
3

асинхронный код

Команды асинхронного кода выполняются по мере получения результатов:

```
console.log('1');  
setTimeout(function afterTwoSeconds() {  
  console.log('2');  
}, 2000)  
console.log('3');
```

коллбэк – функция
обратного вызова

Результат:

1
3
2

*Время получения
результата часто
непредсказуемо.*

АСИНХРОННЫЙ ВЫЗОВ ФУНКЦИИ С ИСПОЛЬЗОВАНИЕМ КОЛЛБЭКА

Допустим у нас есть функция, медленно складывающая 2 числа (скажем, на сервере).

После завершения операции она будет вызывать функцию-коллбэк, передавая ей результат:

```
function add(x,y,f) {  
    setTimeout(()=>f(x+y), 1000);  
}
```

Тогда, чтобы сложить 1+2, потом к результату прибавить 3, и потом его вывести, надо написать такой код:

```
add(1,2,  
    res=>add(res,3,  
        res=>console.log(res)));
```

ИСПОЛЬЗОВАНИЕ THIS В КОЛЛБЭКАХ

```
arr = [1,2,3];

arr.summarize = function() {

  this.sum = 0;

  this.forEach(function(e) { this.sum = this.sum+e; } );

  // В коллбэке this указывает на window, а не на объект arr

}
```

```
console.log(arr.sum); // 0
```

```
console.log(sum); // 6 – потому что this.sum – это window.sum
```

обход проблемы (устаревший подход):

```
arr.summarize = function() {

  this.sum = 0;

  var self = this;

  this.forEach(function(e) { self.sum = self.sum+e; } ); // self попадает в замыкание и продолжает указывать на this

}
```

ИСПОЛЬЗОВАНИЕ THIS В КОЛЛБЭКАХ

```
arr = [1,2,3];

arr.summarize = function() {

    this.sum = 0;

    this.forEach(function(e) { this.sum = this.sum+e; } );

    // В коллбэке this указывает на window, а не на объект arr

}
```

```
console.log(arr.sum); // 0
```

```
console.log(sum); // 6 – потому что this.sum – это window.sum
```

более современный подход – используем bind:

```
this.forEach(function(e) { this.sum = this.sum+e; }.bind(this) );
```

```
// за счет bind мы привязываем функцию к this – теперь this указывает на arr
```

ИСПОЛЬЗОВАНИЕ THIS В КОЛЛЕБАХ

Еще одно решение – использовать стрелочную функцию: в ней **this** продолжает указывать на **arr**:

```
arr.summarize = function() {  
  this.sum = 0;  
  this.forEach(e=>{ this.sum = this.sum+e; });  
}
```


ИСПОЛЬЗОВАНИЕ THIS В КОЛЛБЭКАХ

Рассмотрим пример – вывод кастомизированного сообщения по клику:

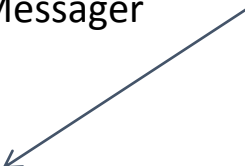
```
class Messenger {  
  constructor(message) {  
    this.message = message;  
  }  
  
  handleClick () {  
    console.log(this.message); // напечатает undefined: this указывает на window:  
                                // метод – обычная функция  
  }  
  
  addClickHandler() {  
    window.onclick = this.handleClick;  
  }  
}  
  
new Messenger("hello from Messenger").addClickHandler();
```

ИСПОЛЬЗОВАНИЕ THIS В КОЛЛБЭКАХ

Решение 1: используем bind(this):

```
class Messenger {  
  constructor(message) {  
    this.message = message;  
  }  
  
  handleClick () {  
    console.log(this.message); // hello from Messenger  
  }  
  
  addClickHandler() {  
    window.onclick = this.handleClick.bind(this);  
  }  
}  
new Messenger("hello from Messenger").addClickHandler();
```

при передаче ссылки
привязываем метод к this с
помощью bind()



ИСПОЛЬЗОВАНИЕ THIS В КОЛЛБЭКАХ

Решение 2: используем стрелочную функцию при привязке:

```
class Messenger {  
  constructor(message) {  
    this.message = message;  
  }  
  
  handleClick () {  
    console.log(this.message); // hello from Messenger  
  }  
  
  addClickHandler() {  
    window.onclick = ()=>this.handleClick();  
  }  
}  
new Messenger("hello from Messenger").addClickHandler();
```

оборачиваем вызов в
стрелочную функцию – this
не теряется

ИСПОЛЬЗОВАНИЕ THIS В КОЛЛБЭКАХ

Решение 3: используем свойство вместо метода:

```
class Messenger {  
  constructor(message) {  
    this.message = message;  
    this.handleClick = ()=>console.log(this.message);  
  }  
  
  addClickHandler() {  
    window.onclick = this.handleClick;  
  }  
}  
  
new Messenger("hello from Messenger").addClickHandler();
```

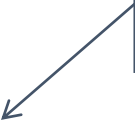
метода нет – объявляем свойство
как ссылку на стрелочную
функцию – тогда проблемы
потери this нет

ИСПОЛЬЗОВАНИЕ THIS В КОЛЛБЭКАХ

Решение 4: используем переприсваивание свойства:

```
class Messenger {  
  constructor(message) {  
    this.message = message;  
    this.handleClick = this.handleClick.bind(this);  
  }  
  
  handleClick () {  
    console.log(this.message); // hello from Messenger  
  }  
  
  addClickHandler() {  
    window.onclick = this.handleClick;  
  }  
}  
  
new Messenger("hello from Messenger").addClickHandler();
```

подменяем ссылку handleClick
на ссылку, привязанную к this



ПРОМИСЫ

В ES2015 появился объект, который содержит будущий результат – Promise – обещание.

Вместо вызова коллбэка мы возвращаем промис.

На момент получения в нем еще нет результата.

Но мы можем дождаться результата, используя функцию then и передав в нее обработчик результата.

```
function add(x,y) {  
  return new Promise(function(resolve) {  
    setTimeout( ()=> resolve(x+y), 1000);  
  });  
}
```

В результате код, использующий промисы вместо коллбэков, выглядит более аккуратно:

```
add(1,2)  
  .then(x=>add(-5,x))  
  .then(res=>console.log(`result = ${res}`))
```

ПРОМИСЫ: ОБРАБОТКА ИСКЛЮЧЕНИЙ

Возможны ситуации, при которой асинхронная функция отрабатывает некорректно.

Например, если это обращение к серверу – возможно сервер не отвечает или отвечает с ошибкой.

Как быть в такой ситуации? Что вернуть?

На этот случай в Promise есть второй аргумент: `reject` - отказ.

Вызывая его, мы говорим: что-то пошло не так, клиент должен обработать эту ситуацию – и вызываем `reject`, передавая туда подробности произошедшего (можно передать любой объект).

```
function add(x,y) {  
  return new Promise(function(resolve,reject) {  
    setTimeout(()=>x>0?resolve(x+y):reject("x should be >0"), 1000);  
  });  
}
```

Клиент, получая промис, может определить, как обработать ошибочную ситуацию, в методе `catch()`:

```
add(1,2) // здесь пока все нормально  
  .then(x=>add(-5,x)) // здесь что-то пошло не так – дальше не идем, прыгаем в catch  
  .then(res=>console.log(`result = ${res}`))  
  .catch(err=>console.log("ERROR:"+err)); // выводим сообщение об ошибке
```

PROMISE.ALL

Бывает ситуация, когда надо запустить на выполнение сразу несколько операций.

Например, нужно скачать с сервера сразу 3 документа. Тогда лучше их запускать всех сразу, а не последовательно. Для этого можно использовать `Promise.all`, передавая в него список промисов:

```
Promise.all([add(1,2),add(2,3),add(5,5)])  
  .then(res=>console.log(res))
```

> [3, 5, 10] (выведется через секунду)

СИНТАКСИС ASYNC/AWAIT

В ES2018 появился новый синтаксис – `async/await`.

Он позволяет еще лаконичнее работать с асинхронным кодом, как будто это синхронный код.

```
function add(x,y) {  
  return new Promise(function(resolve) {  
    setTimeout(()=>resolve(x+y), 1000);  
  });  
}  
  
async function main() {  
  var res = await add(1, 2);  
  var res2 = await add (res, 3);  
  console.log( res2 ); //6  
}  
main();
```

СИНТАКСИС ASYNC/AWAIT : ОБРАБОТКА ИСКЛЮЧЕНИЙ

```
function add(x,y) {  
  return new Promise(function(resolve,reject) {  
    setTimeout(()=>x>0?resolve(x+y):reject("x should be >0"), 1000);  
  });  
}
```

```
async function main() {  
  try {  
    var res = await add(1, 2);  
    var res2 = await add (res, 3);  
    console.log( res2 ); //6  
  } catch(e) {  
    console.log("ERROR:"+e);  
  }  
}  
main();
```

Важное ограничение:
await может использоваться
только внутри **async** функции!

СИНТАКСИС ASYNC/AWAIT : ОГРАНИЧЕНИЯ

```
function add(x,y) {  
  return new Promise(function(resolve,reject) {  
    setTimeout(()=>x>0?resolve(x+y):reject("x should be >0"), 1000);  
  });  
}
```

Вы не можете здесь
использовать async или
await!

```
async function main() {  
  try {  
    var res = await add(1, 2);  
    var res2 = await add (res, 3);  
    console.log( res2 ); //6  
  } catch(e) {  
    console.log("ERROR:"+e);  
  }  
}
```

Важное ограничение:

await может использоваться только внутри
async функции!

```
main();  
console.log("FINISHED");
```

В данном случае мы не ждем результата работы.
FINISHED будет выведено ДО результата вычисления – команда
main() запускает асинхронную функцию, но не ждет ее
завершения!

СИНТАКСИС ASYNC/AWAIT : ОГРАНИЧЕНИЯ

```
function add(x,y) {  
  return new Promise(function(resolve,reject) {  
    setTimeout(()=>x>0?resolve(x+y):reject("x should be >0"), 1000);  
  });  
}
```

Вы не можете здесь
использовать async или
await

```
async function main() {  
  try {  
    var res = await add(1, 2);  
    var res2 = await add (res, 3);  
    console.log( res2 ); //6  
  } catch(e) {  
    console.log("ERROR:"+e);  
  }  
}
```

Важное ограничение:

await может использоваться только внутри
async функции!

```
await main();  
console.log("FINISHED");
```

Может быть так?

1

2

СИНТАКСИС ASYNC/AWAIT : ОГРАНИЧЕНИЯ

```
function add(x,y) {  
  return new Promise(function(resolve,reject) {  
    setTimeout(()=>x>0?resolve(x+y):reject("x should be >0"), 1000);  
  });  
}
```

Вы не можете здесь
использовать `async` или
`await`

```
async function main() {  
  try {  
    var res = await add(1, 2);  
    var res2 = await add (res, 3);  
    console.log( res2 ); //6  
  } catch(e) {  
    console.log("ERROR:"+e);  
  }  
}
```

Важное ограничение:

`await` может использоваться только внутри
`async` функции!

```
await main();  
console.log("FINISHED");
```

Так нельзя! Если внешняя функция не `async`.

СИНТАКСИС ASYNC/AWAIT : ОГРАНИЧЕНИЯ

```
function add(x,y) {  
  return new Promise(function(resolve,reject) {  
    setTimeout(()=>x>0?resolve(x+y):reject("x should be >0"), 1000);  
  });  
}
```

Вы не можете здесь
использовать async или
await

```
async function main() {  
  try {  
    var res = await add(1, 2);  
    var res2 = await add (res, 3);  
    console.log( res2 ); //6  
  } catch(e) {  
    console.log("ERROR:"+e);  
  }  
}
```

Важное ограничение:

await может использоваться только внутри
async функции!

```
main().then(()=>  
  console.log("FINISHED"));
```

А вот так можно.

main() возвращает Promise.

Поэтому можно использовать then() и catch() в синхронном коде.
FINISHED будет напечатан ПОСЛЕ вывода результата.

ФУНКЦИЯ FETCH

```
let promise = fetch(url[, options]);
```

```
fetch('/user/1')
```

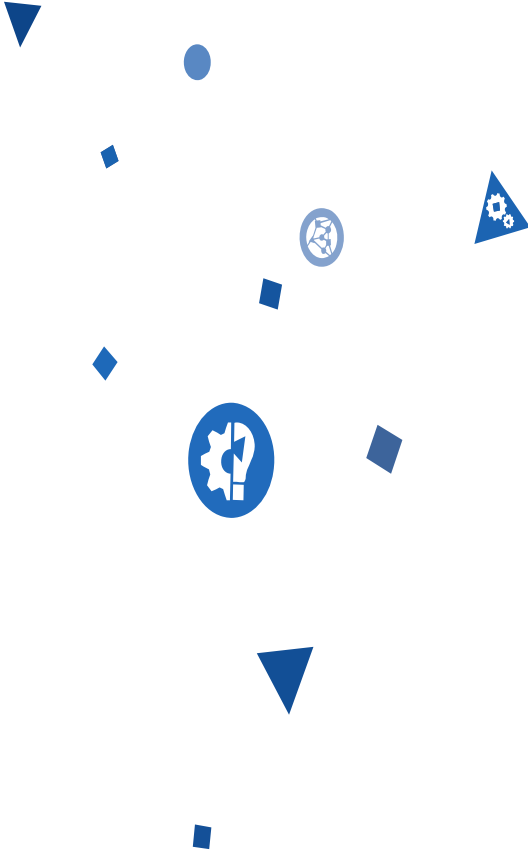
```
.then(function(response) {  
  console.log(response.headers.get('Content-Type'));  
  // application/json; charset=utf-8  
  console.log(response.status); // 200
```

```
  return response.json();  
})  
.then(function(user) {  
  console.log(user.name); // Ivan  
})  
.catch( console.log );
```

Объект **response** кроме доступа к заголовкам **headers**, статусу **status** и некоторым другим полям ответа, даёт возможность прочитать его тело, в желаемом формате.

Варианты включают в себя:

```
response.arrayBuffer()  
response.blob()  
response.formData()  
response.json()  
response.text()
```



МОДУЛЬНОСТЬ В JS СБОРЩИК WEBPACK МЕНЕДЖЕР ПАКЕТОВ NPM

МОДУЛИ JAVASCRIPT

Зачем нужны модули?

Когда приложение сложное и кода много – мы пытаемся разбить его на файлы. В каждом файле описываем какую-то часть, а в дальнейшем – собираем эти части воедино.

Что такое модуль?

Модулем считается файл с кодом.

В этом файле ключевым словом `export` помечаются переменные и функции, которые могут быть использованы снаружи.

Другие модули могут подключать их через вызов `import`.

ЭКСПОРТ ИЗ МОДУЛЯ

Ключевое слово **export** можно ставить:

- перед объявлением переменных, функций и классов
- отдельно, при этом в фигурных скобках указывается, что именно экспортируется

```
// экспорт прямо перед объявлением  
export let one = 1;
```

Можно написать **export** и отдельно от объявления:

```
let two = 2;  
export {two};
```

Для двух переменных будет так:

```
export {one, two};
```

При помощи ключевого слова **as** можно указать, что переменная **one** будет доступна снаружи (экспортирована) под именем **once**, а **two** – под именем **twice**:

```
export {one as once, two as twice};
```

ЭКСПОРТ ФУНКЦИЙ И КЛАССОВ

Экспорт функций и классов выглядит так же:

```
export class User {  
  constructor(name) {  
    this.name = name;  
  }  
};
```

```
export function sayHi() {  
  alert("Hello!");  
};
```

```
// отдельно от объявлений было бы так:  
// export {User, sayHi}
```

```
// функция без имени – так будет ошибка:  
export function() { alert("Error"); };
```

ИМПОРТ МОДУЛЕЙ

Другие модули могут подключать экспортированные значения при помощи ключевого слова `import`.

Синтаксис:

```
import {one, two} from "./nums";
```

Здесь:

- `./nums` – модуль, как правило это путь к файлу модуля.
- `one, two` – импортируемые переменные, которые должны быть обозначены в `nums` словом `export`.

В результате импорта появятся локальные переменные `one, two`, которые будут содержать значения соответствующих экспортов.

Импортировать можно и под другим именем, указав его в «**as**»:

```
// импорт one под именем item1, а two – под именем item2  
import {one as item1, two as item2} from "./nums";
```

```
alert( `${item1} and ${item2}` ); // 1 and 2
```

ИМПОРТ ВСЕХ ЗНАЧЕНИЙ СРАЗУ

Можно импортировать все значения сразу в виде объекта вызовом **import * as obj**, например:

```
import * as numbers from "./nums";
```

```
// теперь экспортированные переменные - свойства  
numbers
```

```
alert( `${numbers.one} and ${numbers.two}` ); // 1 and 2
```

ЭКСПОРТ ПО УМОЛЧАНИЮ

Как правило, код стараются организовать так, чтобы каждый модуль делал **одну вещь**. Иначе говоря, «**один файл – одна сущность**», которую он описывает». Например, файл **user.js** содержит **class User**, файл **login.js** – функцию **login()** для авторизации, и т.п.

При этом модули, разумеется, будут использовать друг друга. Например, **login.js**, скорее всего, будет импортировать класс **User** из модуля **user.js**.

Для такой ситуации, **когда один модуль экспортирует одно значение**, предусмотрено особое ключевое сочетание **export default**. Если поставить после **export** слово **default**, то значение станет «**экспортом по умолчанию**». Такое значение можно импортировать без фигурных скобок.

Например, файл **user.js**:

```
export default class User {  
  constructor(name) {  
    this.name = name;  
  }  
};
```

...а в файле **login.js**:

```
import User from './user';  
new User("Вася");
```

```
// если бы user.js содержал  
export class User { ... }
```

```
// ...то при импорте User понадобились бы фигурные скобки:  
import {User} from './user';  
new User("Вася");
```

РЕ-ЭКСПОРТ

Допустим, у нас есть библиотека, содержащая модуль **math.js**:

```
export function sum (x, y) { return x + y }
```

```
export var pi = 3.141593;
```

Мы не можем менять библиотечный модуль, но нам бы хотелось его изменить или расширить.

Для этого мы делаем модуль **mathplusplus.js**, который ре-экспортирует все возможности **math.js**, расширяя их:

```
// lib/mathplusplus.js  
export * from "math"  
export var pi = 3.1415926;  
export default (x) => Math.exp(x)
```

Теперь, подключая **mathplusplus**, у нас есть все, что было в модуле **math** плюс то, что мы добавили или переопределили:

```
// someApp.js  
import exp, { pi } from "lib/mathplusplus"  
console.log("eπ = " + exp(pi))
```

МЕНЕДЖЕР ПАКЕТОВ NPM

- ◆ Node Package Manager (NPM) - менеджер пакетов Node
- ◆ NPM предоставляет следующие возможности:
 - Онлайн-репозитории для пакетов/модулей Node.js
 - Утилита командной строки для установки пакетов Node.js packages, менеджмента версий и зависимостей

PACKAGE.JSON

Для управления установленными библиотеками используется package.json.

Начальную версию можно создать с помощью **npm init**.

Далее при установке пакетов командой `npm install <package_name>`

Библиотека автоматически добавляется в package.json

Пример **package.json**:

```
{  
  "name": "async-lib",  
  "version": "1.1.2",  
  "description": "Async library",  
  "main": "index.js",  
  "scripts": {  
    "test": "mocha test.js"  
  },  
  "author": "Vladimir Sonkin",  
  "license": "ISC",  
  "keywords": "async",
```

ВЕРСИОНИРОВАНИЕ В NPM

Пример `package.json` (продолжение):

```
"dependencies": {  
  "bluebird": "^3.5.0"  
},  
"devDependencies": {  
  "mocha": "~2.1.0"  
}  
}
```

патч

мажорная версия

минорная версия

^ означает, что возможно обновление до последней минорной версии
например ^3.5.0 может обновиться до 3.7.1, но не до 4.*

~ означает, что возможно обновление до последнего патча
например, ~2.1.0 может обновиться до 2.1.3, но не до 2.2.*

Мажорная версия – значительные изменения библиотеки, без гарантии обратной совместимости

Минорная версия – незначительные изменения, с гарантией обратной совместимости

Патч – исправление ошибок

КОМАНДЫ NPM

- ♦ Для примера используется пакет `express` – но это может быть любая доступная библиотека.
- `npm init` – создать `npm` в диалоговом режиме
- `npm install` – установить все зависимости из `package.json`
- `npm install express` – установить `express`: скачать его в папку `node_modules`, добавить запись в `package.json`
- `npm uninstall express` – удалить `express`
- `npm install express@3.2.1` - установить `express` определенной версии
- `npm search express` – найти `express` в репозитории
- `npm update express` – обновить `express` до последней минорной версии
- `npm install webpack -g` – установить `webpack` глобально
- `npm adduser` – добавить пользователя `npm`
- `npm publish` – опубликовать собственную библиотеку

ПАПКИ NPM

- ◆ **Локальная инсталляция:** помещает все в папку `./node_modules` в корне проекта
- ◆ **Глобальная инсталляция** (с флагом `-g`): помещает в папку `/usr/local` или в папку, где установлен Node
- ◆ Установите пакет локально, если вы собираетесь его импортировать.
- ◆ Установите пакет глобально, если вы собираетесь запускать пакет из командной строки.
- ◆ Если вам нужно и то, и другое, придется установить дважды (альтернатива – использовать `npm link`).

PACKAGE-LOCK.JSON

- ♦ `package-lock.json` автоматически генерируется для любых операций, где npm изменяет либо дерево `node_modules`, либо `package.json`
- ♦ Этот файл предназначен для передачи в системы контроля версий (git, svn) и выполняет различные задачи:
- Описать единое представление дерева зависимостей
- Предоставьте пользователям возможность «путешествовать по времени» в предыдущие состояния `node_modules` без необходимости фиксировать сам каталог.
- Для облегчения большей видимости изменений дерева с помощью читаемых различий в управлении версиями.
- ♦ Таким образом, не нужно сохранять `node_modules` в GIT!
- ♦ `package-lock.json` позволяет однозначно восстановить содержимое папки `node_modules`

ИМПОРТ ИЗ NODE_MODULES И ИЗ ЛОКАЛЬНОГО ФАЙЛА

При импорте возможно 2 варианта синтаксиса:

```
import {one} from "./nums";  
или  
import {one} from "nums";
```

Первый вариант – импорт из локального файла. Здесь будет найден файл относительно текущего файла (в той же папке) и импортирован.

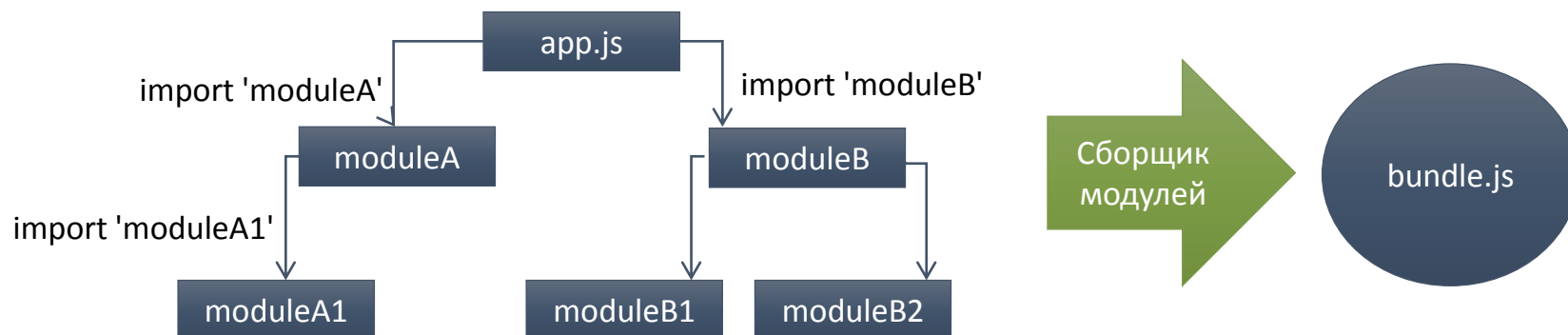
Второй вариант – импорт из папки node_modules. Будет найдена папка nums, в ней – файл index.js и его содержимое будет импортировано.

РАБОТА С МОДУЛЯМИ В БРАУЗЕРЕ

Однако поддержка модулей не реализована в браузерах.

Причина – загрузка модулей по одному приведет к очень долгой загрузке сайта: ведь таких модулей, вместе со всеми требуемыми библиотеками, может быть тысячи!

Что же делать? Можно использовать **сборщик модулей**:



На выходе мы получаем 1 файл, который можно подключить в SPA (одностраничное приложение):

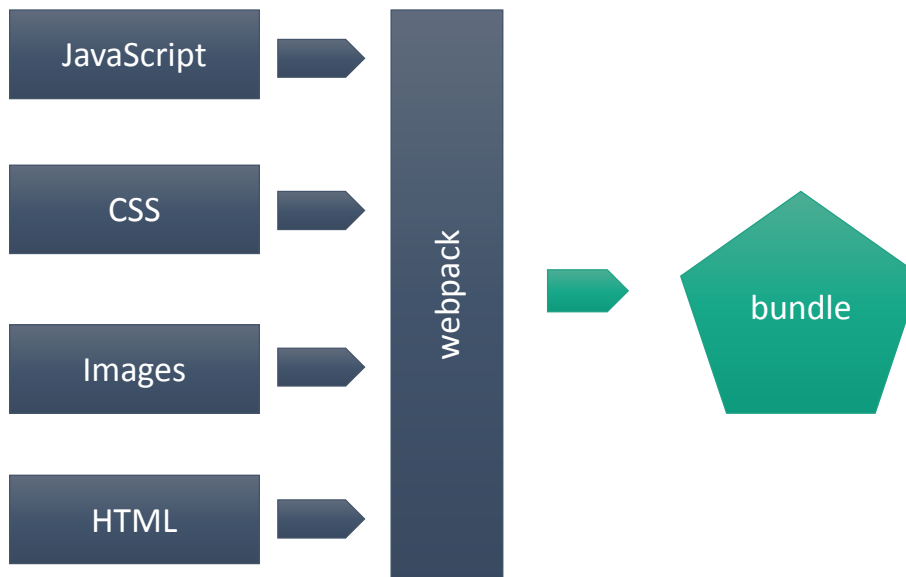
```
<script src="bundle.js"></script>
```

При этом он будет включать **все** необходимые **модули** и **все библиотеки**!

ЧТО ТАКОЕ WEBPACK?

Самый популярный сборщик модулей – это Webpack.

Webpack берет модули с зависимостями и генерирует статические ресурсы, которые представляют эти модули.



Когда webpack обрабатывает ваше приложение, он создает граф зависимостей, который отображает каждый модуль, необходимый вашему проекту, и генерирует один или несколько пакетов (bundle).

КАК НАЧАТЬ РАБОТУ С WEBPACK?

Для установки вебпака нужно установить Node.JS

После этого можно установить вебпак глобально:

```
$ npm install -g webpack
```

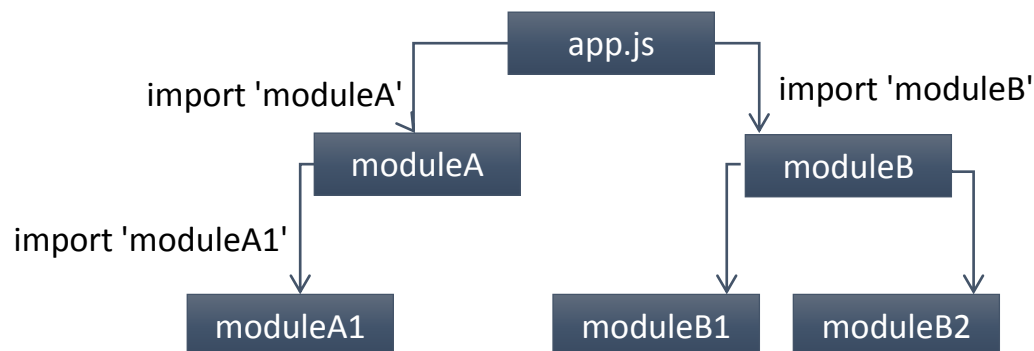
ТОЧКА ВХОДА

Точка входа указывает, какой модуль webpack должен использовать, чтобы начать строить свой внутренний граф зависимостей. Webpack будет определять, какие из других модулей и библиотек зависят от точки входа (прямо и косвенно).

По умолчанию его значение равно `./src/index.js`, но вы можете указать другую (или несколько точек входа), настроив свойство записи в конфигурации webpack. Например:

webpack.config.js:

```
module.exports = {  
  entry: './app.js'  
};
```



ВЫХОДНОЙ ФАЙЛ

Свойство `output` указывает webpack, где следует испускать пакеты, которые он создает, и имена этих файлов. По умолчанию используется `./dist/main.js` для основного выходного файла и папки `./dist` для любого другого сгенерированного файла.

Вы можете настроить эту часть процесса, указав поле вывода в своей конфигурации:

```
const path = require('path');

module.exports = {
  entry: './app.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'my-first-webpack.bundle.js'
  }
};
```

РЕЖИМЫ WEBPACK

Установив параметр режима как для разработки - `development`, или для конечного клиента - `production`, вы можете включить встроенные оптимизации webpack, соответствующие каждой среде. Значение по умолчанию - это `production`:

```
module.exports = {  
  mode: 'production'  
};
```

Также режим может быть задан при запуске Webpack в CLI:

```
webpack --mode=production
```

Настройки режима **development**:

- устанавливает значение переменной `process.env.NODE_ENV` в значение **development**
- включает плагины `NamedChunksPlugin` и `NamedModulesPlugin`

Настройки режима **production**:

- устанавливает значение переменной `process.env.NODE_ENV` в значение **production**
- включает плагины `FlagDependencyUsagePlugin`, `FlagIncludedChunksPlugin`, `ModuleConcatenationPlugin`, `NoEmitOnErrorsPlugin`, `OccurrenceOrderPlugin`, `SideEffectsFlagPlugin` и `UglifyJsPlugin`

ЗАГРУЗЧИКИ

Из коробки webpack понимает только файлы JavaScript и JSON. Загрузчики позволяют webpack обрабатывать файлы других типов и преобразовывать их в допустимые модули, которые могут быть использованы вашим приложением и добавлены в граф зависимостей.

```
module.exports = {  
  output: {  
    filename: 'my-first-webpack.bundle.js'  
  },  
  module: {  
    rules: [  
      { test: /\.txt$/, use: 'raw-loader' }  
    ]  
  }  
};
```

ПРИМЕРЫ ЗАГРУЗЧИКОВ

Вы можете использовать загрузчики, чтобы сообщить webpack, как загрузить файл CSS или преобразовать TypeScript в JavaScript. Для этого вы должны начать с установки необходимых вам загрузчиков:

```
npm install --save-dev css-loader
```

```
npm install --save-dev ts-loader
```

Затем сконфигурируйте webpack использовать css-загрузчик для каждого .css-файла и загрузчик ts-loader для всех файлов .ts:

```
module.exports = {  
  module: {  
    rules: [  
      { test: /\.css$/, use: 'css-loader' },  
      { test: /\.ts$/, use: 'ts-loader' }  
    ]  
  }  
};
```

КОНФИГУРИРОВАНИЕ ЗАГРУЗЧИКОВ

`module.rules` позволяет указать несколько загрузчиков в конфигурации вашего веб-пакета. Это краткий способ отображения загрузчиков и помогает поддерживать чистый код. Он также предлагает вам полный обзор каждого соответствующего загрузчика.

Загрузчики выполняются, начиная с последнего. В приведенном ниже примере выполнение начинается с `sass-loader`, продолжается `css-loader` и, наконец, заканчивается загрузчиком стилей:

```
module.exports = {  
  module: {  
    rules: [  
      {  
        test: /\.css$/,  
        use: [  
          { loader: 'style-loader' },  
          {  
            loader: 'css-loader',  
            options: {  
              modules: true  
            }  
          },  
          { loader: 'sass-loader' }  
        ]  
      }  
    ]  
  }  
};
```

ВОЗМОЖНОСТИ ЗАГРУЗЧИКОВ

- Загрузчики могут быть выстроены в **цепочки**. Каждый загрузчик в цепочке применяет преобразования к обрабатываемому ресурсу. Цепочка выполняется в **обратном порядке**. Первый загрузчик передает свой результат (ресурс с прикладными преобразованиями) на следующий и т. д. Webpack ожидает, что **в результате** выполнения всей цепочки будет получен **JavaScript**.
- Загрузчики могут быть синхронными или асинхронными.
- Загрузчики работают в **Node.js** и могут делать все, что может Node.
- Загрузчики могут быть настроены с помощью объекта **options**.
- **Плагины** могут предоставить загрузчикам больше возможностей.
- Загрузчики могут создавать **дополнительные файлы**.