# Блок 3. EcmaScript 2015.

## Занятие 1. Модули в JavaScript. Работа с WebPack.

Мы написали уже достаточно большую программу, но порядка в ней нет: куча функций, отовсюду можно вызвать любую функцию.

Мы заранее аккуратно разделили код на отдельные модули ui.js и service.js - модуль интерфейсного слоя и бизнес-логику, но пока что у нас отсутствует инкапсуляция - сокрытие частей программы, не нужных другим слоям. Именно этим мы займемся, используя WebPack и модули JavaScript.

### 1) Разделение на модули

Установите webpack:

**npm install -g webpack-cli**

**npm install -g webpack**

Создайте файл **webpack.config.js** со следующим содержанием:

module.**exports** = {  
 **entry**: **'./main.js'**,  
 **devtool**: **'source-map'**,  
  
 **output**: {  
 **filename**: **'./bundle.js'** }  
};

Поместите все файлы .js в папку **employees**.

В файле employees-json.js экспортируйте данные:

**export const** DATA = { ... }

Теперь добавьте импорт в файл **service.js**:

**import** { DATA } **from './employees-json'**;

Из service.js экспортируйте функции, которые нужны в ui.js, например:

**export function** *getEmployees*() { ... }

К таким функциям можно отнести addEmployee, removeEmpoloyee, findById, searchEmployees, setEmployeeManager.

Теперь импортируйте все эти функции в **ui.js**:

**import** { *getEmployees*, *removeEmployee*, *addEmployee*,  
 *findById*, *searchEmployees*, *setEmployeeManager*} **from './service'**;

Из **ui.js** в свою очередь надо экспортировать функции, которые понадобятся в интерфейсе - в **index.html**: это функции runUI, addEmployeeUI, openTab, searchEmployeeUI.

Создайте файл main.js и импортируйте в него то, что было экспортировано из **ui.js**:

**import** { *runUI*, *addEmployeeUI*, *openTab*, *searchEmployeeUI* } **from './employees/ui'**;

В **main.js** мы должны сделать функции доступными извне - они будут вызываться из **index.html**. Для того, чтобы сделать это, надо присвоить их глобальному объекту window. Также надо запустить runUI().

***window***.**addEmployeeUI** = *addEmployeeUI*;  
***window***.*openTab* = *openTab*;  
***window***.**searchEmployeeUI** = *searchEmployeeUI*;

runUI();

или так:

***window***.addEventListener(**"load"**, runUI);

Теперь можно собрать весь код с помощью webpack. Для этого запустите webpack в папке, в которой находится **webpack.config.js**

В результате появится папка **dist**, и в ней - файл **bundle.js**, содержащий весь собранный код.

Остается подключить этот код к проекту. Для этого в конец index.html вместо тех скриптов, которые подключались ранее, добавим следующее:

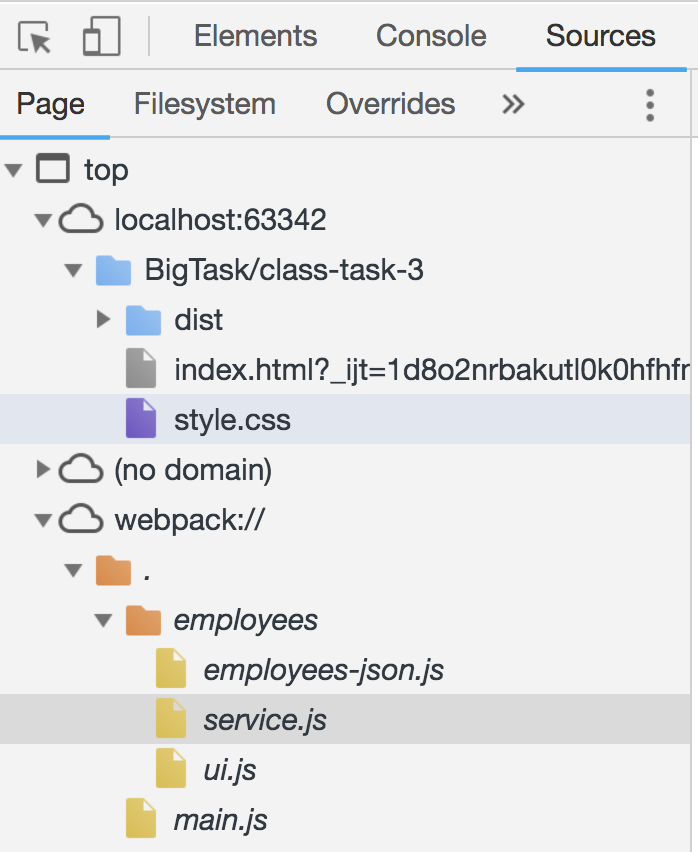
<**script src="./dist/bundle.js"**></**script**>

Теперь у нас есть четкое разграничение областей видимости и разделение программы по уровням:

представление (index.html) -> логика представления (ui.js) ->бизнес-логика (service.js) -> данные (employees-json.js).

Такой подход делает код значительно более управляемым, но мы только на первом шаге улучшения кода.

### 2) Отладка с использованием source-map



Также обратите внимание, что мы добавили в webpack.confing.js строчку

**devtool**: **'source-map'**

Таким образом мы включаем генерацию source map. Эта возможность позволяет связать код **bundle.js,** который выполняется в браузере, с кодом в отдельных модулях, и позволяет видеть в отладчике отдельные модули. Откройте отладчик и найдите исходные файлы модулей (см. иллюстрацию).

Поставьте точки останова и попробуйте поработать с отладчиком.

### 3) Загружаем CSS через WebPack.

CSS-файлы тоже можно включать в bundle.js и загружать через WebPack. Для этого нам понадобится установить CSS-загрузчик:

**npm install --save-dev style-loader css-loader**

Также необходимо модифицировать **webpack.config.js**, добавив в него описание загрузчика **CSS**:

**output**: {  
 **filename**: **'./bundle.js'**},  
  
**module**: {  
 **rules**: [  
 {  
 **test**: /\.css$/,  
 **use**: [  
 **'style-loader'**,  
 **'css-loader'** ]  
 }  
 ]  
 }

Теперь мы можем импортировать CSS напрямую в main.js:

**import './style.css'**;

При этом загрузку CSS в **index.html** можно больше не использовать, удалив строчку

<**link rel="stylesheet" type="text/css" href="style.css"**>

Перезагрузите код, чтобы убедиться, что все работает так же.

### 4) Экспорт из модуля по умолчанию

ES2015 поддерживает синтаксис экспорта и импорта по умолчанию.

Изменим код **employees-json.js** c

**export const** DATA = { ... }

на

**export default** { ... }

Также изменим импорт в файле service.js c

**import {** DATA } **from './employees-json'**;

на

**import** DATA **from './employees-json'**;

Такой экспорт по умолчанию в файле может быть только один. Экспортировать можно объект (как в данном случае), функцию, массив, любое значение, класс и так далее.

Как мы видим, при экспорте имя не задается, и устанавливается при импорте - так, в данном случае имя могло быть не DATA, а любым другим.

В случае, если в файле есть дефолтный экспорт, при импорте он должен указываться без фигурных скобок, при этом все остальные импорты должны быть в фигурных скобках.

## Занятие 3. Объектно-ориентированное программирование в ES2015

### 1) Создаем классы Person и Employee

Создадим классы Person и Employee в папке model, в файлах Person.js и Employee.js:

**class** Person {

}

**class** Employee extends Person {

}

Также сделайте эти классы экспортируемыми.

Реализуйте конструктор для класса Person:

constructor(name, surname) {  
 **this**.**name** = name;  
 **this**.**surname** = surname;  
}

В классе Employee нам тоже понадобится аналогичный конструктор:

constructor(name, surname, department) {  
 **super**(name,surname);

**this**.**department** = department;  
}

Данный конструктор вызывает конструктор родительского класса, но также может выполнять дополнительную инициализацию - в данном случае позволяет установить департамент.

В классе Person реализуем геттер для полного имени:

**get** fullName() {  
 **return this**.**name**+**" "**+**this**.**surname**;  
}

А также для вычисления возраста (перенесем из service.js и немного модифицируем):

**get** age() {  
 **if** (!**this**.**\_dateOfBirth**) **return ""**;  
 **let** ageDiff = ***Date***.now() - **this**.**\_dateOfBirth**.getTime();  
 **let** ageDate = **new *Date***(ageDiff); *// miliseconds from epoch* **return " <b>Возраст:</b>"**+

***Math***.abs(ageDate.getUTCFullYear() - 1970);  
}

Также нам понадобится метод formatDate (перенесем из service.js)::

formatDate(date) {  
 **let** day = date.getDate();  
 **if** (day<10) day = **'0'**+day;  
 **let** month = date.getMonth()+1;  
 **if** (month<10) month = **'0'**+month;  
 **let** year = date.getFullYear();  
 **return** day + **'.'** + month + **'.'** + year;  
}

Также реализуем методы для чтения и записи даты рождения. Будем устанавливать дату, получая ее из строки, а возвращать в отформатированном виде:

**set** dateOfBirth(date) {  
 **this**.**\_dateOfBirth** = **new *Date***(date);  
}  
  
**get** dateOfBirth() {  
 **return this**.**\_dateOfBirth**?  
 **" <b>Дата рождения:</b> "**+**this**.formatDate(**this**.**\_dateOfBirth**):  
 **""**;  
}

Обратите внимание, что мы возвращаем пустую строку в случае, если дата не установлена.

Теперь мы можем перенести код функции getEmployeeInfo() в класс и использовать его для генерации строкового представления класса в классе Person:

toString() {  
 **const** phones = **this**.**phones**?  
 **`Список телефонов:** ${**this**.**phones**}**`**:**''**;  
 **return `** ${**this**.fullName} **\** ${**this**.dateOfBirth} ${**this**.age} ${phones}**`**;  
}

Теперь попробуйте в консоли создать сотрудника с помощью конструктора и вывести его на печать. Для того, чтобы класс Employee был доступен из консоли, нам придется поместить его в глобальный объект window - добавьте это в конец Employee.js:

window.Employee = Employee;

Также надо импортировать класс Employee в main.js:

**import** {Employee} **from './employees/model/Employee'**;

Теперь мы в консоли можем создать объект Employee и вывести его:

e = new Employee("Sveta","Svetova");

console.log(e.toString());

Также мы можем установить дату рождения:

e.dateOfBirth="2000-01-01"

console.log(e.toString());

Теперь будет выведена отформатированная дата и возраст.

Также можно добавить телефон:

e.addPhone("222")

console.log(e.toString());

### 2) Используем новые классы для упаковки объектов JSON

Теперь наша задача - получать объект из имеющегося объекта JSON.

Для этого нам пригодится метод Object.assign(object1, object2), который производит копирование всех свойств и возвращает результирующий объект. Например, вызов

***Object***.assign(**new** Person(), obj)

Создаст объект класса Person, заполненный данными из объекта obj.

У нас нет конструктора без параметров, но в данном случае это неважно - при вызове new Person() поля name и surname получат значение undefined, но потом будут получены из объекта obj. Таким образом, мы можем реализовать следующий метод:

**static** *fromJSON*(obj) {  
 **return *Object***.assign(**new** Person(), obj)  
}

Это - статический метод - фабрика объектов Person. Например, вызов

Person.fromJSON({name: "Михаил", surname:"Михайлов"})

вернет объект класса Person.

Добавьте аналогичный метод также в класс Employee.

Теперь, чтобы получить список всех сотрудников не в виде простых объектов, а упакованных в класс Employee, мы можем реализовать такую функцию в файле Employee.js:

**export function** *jsonToEmployees*(employeesJSON) {  
 **let** employees = [];  
 **for** (**let** e **of** employeesJSON) {  
 employees.push(Employee.*fromJSON*(e));  
 }  
 **return** employees;  
}

В целях тестирования добавьте такую глобальную функцию в Employee.js:

window.allEmployees = function() {

return jsonToEmployees(DATA.employees);

}

Теперь вы можете вывести список всех сотрудников, упакованных в класс Employee, в консоли:

allEmployees()

Попробуйте также получить отчеты обо всех сотрудниках:

allEmployees().join("")

Для того, чтобы в отчетах увидеть дату рождения и возраст сотрудника, добавьте дату в employees-json.js:

{  
 **id**: 1,  
 **name**: **"Пафнутий"**,  
 **surname**: **"Пафнутьев"**,  
 **department**: **"IT"**,  
 **dateOfBirth**: **"2000-01-01"**},

### 3) Используем классы в пользовательском интерфейсе

Осталось воспользоваться новыми классами для вывода данных на экран.

Для этого импортируем класс Employee и функцию jsonToEmployees() в **ui.js**:

**import** {Employee,jsonToEmployees} **from "./model/Employee"**;

В функции showEmployees() модифицируем цикл по employees - теперь цикл будет идти по сотрудникам, упакованным в класс Employee:

**for** (**let** employee **of** jsonToEmployees(employees)) {

...

И теперь для установки значения <li> воспользуемся таким присвоением:

li.innerHTML = employee;

Этот метод вернет подробную информацию о сотруднике.

Если установлена дата рождения, дата и возраст будут отображаться в списке.

Использование классов-упаковок позволяет локализовать логику работы с объектом, а также использовать иерархию объектов и полиморфизм.