Лекция 26. HTTP, Observable, RxJS

[**HTTP**](#_4atxedjxmei3) **3**

[HTTP и взаимодействие с сервером](#_ryhyg7xrsks) 3

[Обработка сложных данных](#_dcqcgf4jkt0t) 4

[Создание сервиса для работы с сервером](#_7caoujrgk4h3) 4

[Работа с массивом данных](#_jawxsg9vm351) 5

[Передача данных в url запроса](#_4a6zqdncnfj3) 6

[Передача объекта на сервер](#_safme68kuv8z) 6

[**RxJS**](#_crburjfzwos4) **8**

[Многопоточность](#_d414erga4alo) 8

[Генераторы](#_37y3w0zhc7yg) 8

[Паттерн Observer (Наблюдатель)](#_rpmh32gd4ve0) 9

[Observable](#_w7rephva0m3r) 10

[Дополнительные возможности RxJS](#_z3ejjbf29r70) 10

[Запросы к серверу с помощью RxJS и Observable](#_lf26oja0cjpb) 10

[Интервал обновления на RxJS](#_yqipi9n32bbp) 12

[**Задание для закрепления материала**](#_uz4x476u6r6r) **14**

[**Дополнительная информация:**](#_8nciqpqrnpt6) **15**

# HTTP

Для взаимодействия с сервером применяется модуль HttpClientModule из пакета "@angular/common/http". Его необходимо установить и импортировать в файле app.module:

import { HttpClientModule } from '@angular/common/http';

@NgModule({

imports: [ BrowserModule, FormsModule, HttpClientModule],

declarations: [ AppComponent],

bootstrap: [ AppComponent ]

})

## HTTP и взаимодействие с сервером

Добавим файл user.json в папку проекта:

{

"name": "Bob",

"age": 28

}

и доработаем app.component:  
  
import { Component, OnInit} from '@angular/core';

import { HttpClient} from '@angular/common/http';

@Component({

selector: 'my-app',

template: `<div>

<p>Имя пользователя: {{user?.name}}</p>

<p>Возраст пользователя: {{user?.age}}</p>

</div>`

})

export class AppComponent implements OnInit {

user: Object;

constructor(private http: HttpClient){}

ngOnInit(){

this.http.get('user.json').subscribe((data:User) => this.user=data);

}

}

## Обработка сложных данных

### Создание сервиса для работы с сервером

В большинстве случаев, обращение к серверу происходит не из компонента, а из отдельного сервиса:

import {Injectable} from '@angular/core';

import {HttpClient} from '@angular/common/http';

@Injectable()

export class HttpService{

constructor(private http: HttpClient){ }

getData(){

return this.http.get('user.json')

}

}

Более того, вполне допустимо использовать отдельные сервисы для каждого типа данных:

* UsersService
* GoodsService
* CatalogueService
* etc.

Это позволяет работать с конкретными данными из конкретного объекта. Что, в свою очередь, позволяет не мешать методы доступа, разграничивать разделы системы.

Теперь мы можем использовать сервис компоненте

import { Component, OnInit} from '@angular/core';

import { HttpService} from './http.service';

@Component({

selector: 'my-app',

template: `<div>

<p>Имя пользователя: {{user?.name}}</p>

<p>Возраст пользователя: {{user?.age}}</p>

</div>`,

**providers: [HttpService]**

})

export class AppComponent implements OnInit {

constructor(private httpService: HttpService){}

ngOnInit(){

this.httpService.getData().subscribe((data) => this.user=data);

}

}

### Работа с массивом данных

В том случае, если необходимо работать с массивом данных, нам потребуется изменить и компонент и сервис:

**users.json:**

{

"userList":

[{

"name": "Bob",

"age": 28

},{

"name": "Tom",

"age": 45

},{

"name": "Alice",

"age": 32

}]

}

**http.service.ts**

import {Injectable} from '@angular/core';

import {HttpClient} from '@angular/common/http';

@Injectable()

export class HttpService{

constructor(private http: HttpClient){ }

getData(){

return this.http.get('users.json')

}

}

**app.component.ts**@Component({

selector: 'my-app',

template: `<ul>

<li \*ngFor="let user of users">

<p>Имя пользователя: {{user?.name}}</p>

<p>Возраст пользователя: {{user?.age}}</p>

</li>

</ul>`,

providers: [HttpService]

})

export class AppComponent implements OnInit {

users = [];

constructor(private httpService: HttpService){}

ngOnInit(){

this.httpService.getData().subscribe(data => this.users=data["userList"]);

}

}

### Передача данных в url запроса

Иногда необходимо передать на сервер какие-либо параметры, проще всего это сделать через url строку :

getUsers(limit: number){

return this.http.get(‘users.json?limit=' + limit);

}

### Передача объекта на сервер

Большие объекты или структуры данных не принято передавать в GET запросе, для этого используется POST метод:

postData(user: User){

const body = {name: user.name, age: user.age};

return this.http.post('http://localhost:60820/api/values', body);

}

# RxJS

RXJS - это огромный набор библиотек, позволяющий оптимизировать очень разные вещи. Основными принципами библиотеки RxJS является многопоточность и генераторы, без понимания этих принципов изучать RxJS и Angular не представляется возможным.

Обязательно изучите:  
<https://www.youtube.com/playlist?list=PLLvTAhHe8AYCmPYhSRNRptBxnXvr5i0eD>

## Многопоточность

Википедия: Многопото́чность — свойство платформы (например, операционной системы, виртуальной машины и т. д.) или приложения, состоящее в том, что процесс, порождённый в операционной системе, может состоять из нескольких потоков, выполняющихся «параллельно», то есть без предписанного порядка во времени. При выполнении некоторых задач такое разделение может достичь более эффективного использования ресурсов вычислительной машины.

Если говорить простым языком, JavaScript позволяет создавать отдельный поток для вычислений или действий.

Самым простым примером применения многопоточности являются функции setTimeout и setInterval. Они позволяют отложить выполнение какого либо кода на определенное время. Слабым местом этих функций является то, что они не способны возвращать значения в основной поток. Т.е. фактически, они способны изменить переменную или добавить что-либо на страницу, но в основной поток они не вернут ничего. Еще одним минусом является то, что нам должно быть известно известно время срабатывания.

Более сложным вариантом мультипоточности являются Promis-ы, они могут возвращать значение в основной поток, позволяют сделать много чего, но и они ограничены:

* Промис, закончивший свое исполнение закрывает поток, и более непригоден для использования
* Промис может вернуть значение лишь единожды.

## Генераторы

Правильнее всего будет воспринимать генератор как «замороженный вызов функции»:

**function\* generator1() {  
 yield {id: 1, title: “Title for news1”};**

**yield {id: 2, title: “Title for news2”};**

**yield {id: 3, title: “Title for news3”};**

**yield {id: 4, title: “Title for news4”};  
}**

При каждом вызове функции генератора (метод генератора .next()) будет возвращаться объект:

**{  
 value: current\_value  
 done: false // true в том случае, если выводится последний объект**

**}**

т.е. мы можем перебирать элементы генератора вызовом **generator1.next()** до тех пор, пока наш список не будет закончен:

**let items = generator1();**

**while( let item = items.next() && item.done !== false ) {  
 console.log(“Generator item: ”, item.value)  
}**

В нашем примере генератор имеет ограниченное количество значений, но это не значит, что эти значения не могут быть предоставлены внешним источником.

## Паттерн Observer (Наблюдатель)

Самое простое объяснение паттерна наблюдатель выглядит следующим образом:

Класс, реагирующий на определенное событие, предоставляет возможность другим классам подписаться на это событие. При наступлении события, уведомляются все объекты / классы из списка подписавшихся.

Самый простой пример:  
**button.addEventListener(‘click’, function(){  
…...  
});**

**button.addEventListener(‘click’, function(){  
…...  
});**

Таким образом, у нас имеется 2 подписавшихся на одно событие, и они оба будут выполнены при клике по кнопке.

## Observable

Observable представляет собой некую помесь реализации паттерна Observer и генератора. Т.е. предоставляет возможность подписаться на каждый “выброс” элемента (yield). Подписанных может быть неограниченное количество.

Основной задачей Observable является унификация работы с любыми источниками данных (в т.ч. и внешними). Т.е. с любым источником (массивом, объектом, запросом к серверу, изменяемым массивом данных) мы можем работать одинаково (полиморфизм).

Подписываясь на изменения в объекте Observable мы получаем результат единожды (если это не событие), или многократно (событие dom элемента, многократно выполняемый запрос)

## Дополнительные возможности RxJS

Помимо работы с любыми типами данных и конвертации их в Observable, RxJS позволяет манипулировать как данными, так и условиями срабатывания событий в Observable.

Примеры манипуляции данными:  
  
 **getNews():Observable<News[]> {**

**return this.http.get<News[]>('assets/news.json').pipe(**

**map(data => data.map(newsItem => new News({title: newsItem.title + ': changed'})))**

**);**

**}**

в данном случае метод pipe позволяет применять к данным обработки любого типа

**map - применяется для каждого элемента, возвращенного генератором**

второй map уже работает как стандартный метод массива, применяя обработку для каждого элемента массива

**filter позволяет фильтровать возвращаемые генератором значения**

и т.д

## Запросы к серверу с помощью RxJS и Observable

Давайте создадим простой компонент latest-news:

import { Component, OnInit } from '@angular/core';

import { NewsService } from '../../services/news.service';

import { Observable } from 'rxjs';

import { News } from '../../models/news';

@Component({

selector: 'news-list',

template:`<ul >

<li \*ngFor="let n of **(news$ | async)**">

{{n.title}}

</li>

</ul>

`,

styleUrls: ['./news-list.component.css']

})

export class NewsListComponent implements OnInit {

**news$:Observable<News[]> = this.NewsSvc.getNews();**

constructor(private NewsSvc: NewsService) { }

ngOnInit() { }

}

Обратите внимание на символ $ в названии атрибута news. В Angular | RxJS | TypeScript существует соглашение, что переменные типа Observable или Promise должны оканчиваться символом $.

Также обратите внимание на шаблон компонента  
(news$ | async) - это говорит о том, что мы подписались на обновление атрибута news$. При этом, как только элемент news$ изменится, то сразу будет произведен перерасчет и будут выведены новые элементы.

**NewsService**

import { Injectable } from '@angular/core';

import { Observable } from 'rxjs';

import { News } from '../models/news';

import { HttpClient } from '@angular/common/http';

@Injectable({

providedIn: 'root'

})

export class NewsService {

constructor( private http:HttpClient ) { }

getNews():Observable<News[]> {

**return this.http.get<News[]>('assets/news.json');**

}

}

тут все на самом деле просто: мы создаем запрос к серверу и возвращаем его результат в виде Observable. Нотации типов были добавлены, чтобы убрать ошибки в IDE.

## Интервал обновления на RxJS

Изменим код компонента LatestNews:

import { Component, OnInit } from '@angular/core';

import { NewsService } from '../../services/news.service';

**import { Observable, timer } from 'rxjs';**

import { News } from '../../models/news';

**import { switchMap } from 'rxjs/operators'**

@Component({

selector: 'news-list',

templateUrl: './news-list.component.html',

styleUrls: ['./news-list.component.css']

})

export class NewsListComponent implements OnInit {

constructor(private NewsSvc: NewsService) { }

**updateNewsTimer$ = timer(0, 3000);**

news$:Observable<News[]> = **this.updateNewsTimer$.pipe(**

**switchMap( () => this.NewsSvc.getNews())**

**);**

ngOnInit() { }

}

В данном примере мы изменили только код компонента, при этом не трогая сервис - достаточно удобно, без каких-либо последствий для остальных компонентов, которые используют тот же самый сервис.

**Как это работает?**

Первым делом мы импортировали функцию timer из библиотеки rxjs.

Далее мы создали атрибут(переменную) контроллера в которую поместили таймер - **updateNewsTimer$ = timer(0, 3000)**, который сработает первый раз при запуске страницы (0 - первый параметр) и будет срабатывать каждые 3 секунду (3000 милисекунд - второй параметр). Обратите внимание, что в названии переменной таймера также присутствует $.

Второе: мы изменили источник для массива (по факту это уже не массив) news$:  
  
 **news$:Observable<News[]> = this.updateNewsTimer$.pipe(**

**switchMap( () => this.NewsSvc.getNews())**

**);**

тем самым сказав, что при каждом срабатывании таймера мы будем вызывать метод getNews() сервиса NewsService.

# 

# Задание для закрепления материала

Вынести конфигурацию приложения(оставить возможность переключать параметры вручную) в отдельный файл, который будет загружаться по http.

Конфигурация должна включать в себя:

* Сложность игры
* размер полей
* количество и размеры кораблей

Файл конфигурации должен быть изменяем и располагаться в папке скомпилированного проекта без дополнительной обработки (обфускация, аглификация)

# Дополнительная информация:

Обязательно:

<https://www.youtube.com/playlist?list=PLLvTAhHe8AYCmPYhSRNRptBxnXvr5i0eD>

Дополнительно:

<https://www.youtube.com/watch?v=3LKMwkuK0ZE>

<https://metanit.com/web/angular2/6.1.php>

<https://habr.com/company/ruvds/blog/341880/>

<https://www.rxworkshop.com/>  
<https://www.youtube.com/watch?v=K7AvXUNB2X8>

<https://www.youtube.com/watch?v=3LKMwkuK0ZE>

<https://www.youtube.com/watch?v=KOOT7BArVHQ>