

Desenvolvimento Web

Trilha JavaScript com Angular e Node

Instrutor: Júlio Pereira Machado (julio.machado@pucrs.br)



Angular e RxJS



Angular - Métodos Assíncronos

- Diferentes opções para implementação:
 - Método recebe um *callback*
 - Método retorna uma *Promise*
 - Método retorna um ***Observable***
- Documentação:
 - <https://angular.io/guide/observables>
 - <https://angular.io/guide/rx-library>
 - <https://angular.io/guide/observables-in-angular>

RxJS

- Reactive Extensions Library for JavaScript
- É uma biblioteca que fornece o conceito de objetos *Observable*
- Implementa o conceito dos padrões *Observer* e *Publish/Subscribe* para comunicação entre objetos

<https://rxjs-dev.firebaseapp.com/>



RxJS

- Conceitos essenciais:
 - Observable – implementa a ideia de uma coleção de valores ou eventos futuros; representa um produtor
 - Observer – é uma coleção de callbacks responsáveis por “observar” os valores entregues pelos objetos Observable; representa um consumidor
 - Subscription – representa a execução de um Observable
 - Subject – representa um tipo especial de Observable que é utilizado para o envio de valores/eventos para múltiplos objetos Observer
- Mais informações:
 - <https://www.learnrxjs.io>

RxJS

- Comparação Promise versus Observable:
 - Observable são objetos declarativos, execução somente ocorre quando há uma subscrição
 - Permite declarar o objeto e usar depois quando necessário
 - Promise são executadas imediatamente
 - Observable permite retornar múltiplos objetos com o passar do tempo (pense num fluxo de dados sendo produzidos)
 - Promise retorna somente um único objeto
 - Observable possui método subscribe() responsável por tratar erros
 - Promise “empurram” erros para a promise “filha”
 - Observable é cancelável
 - Promise não possui esse conceito

RxJS

Retorna Promise

```
doAsyncPromiseThing()  
  .then(() => console.log("I'm done!"))  
  .catch(() => console.log("Error'd out"));
```

Retorna Observable

```
doAsyncObservableThing()  
  .subscribe(  
    () => console.log("I'm done!"),  
    () => console.log("Error'd out")  
  )
```

RxJS

- Funcionamento:

- *Observable* são usualmente criados via método construtor ou *Observable.create()* ou algum operador de criação (*of*, *from*, *interval*, etc)
- A biblioteca RxJS possui uma coleção de diversos “operadores” que manipulam objetos JavaScript que resultam em objetos *Observable* sendo criados
- *Observable* emitem notificações a objetos *Observer* via métodos *next()*, *error()*, *complete()*
 - *Next* – envia um valor; podem ocorrer vários
 - *Error* – envia um objeto *Error* ou exceção; só ocorre uma vez
 - *Complete* – não envia valor; só ocorre uma vez
- Para executar o *Observable* e iniciar o recebimento de notificações, chama-se o método *subscribe(observer)* passando um *Observer* que irá receber as notificações
 - Esse método retorna um objeto *Subscription* que possui um método *unsubscribe()* que é utilizado para parar de receber notificações

RxJS

- Exemplos: criação de um Observable

```
import { Observable } from 'rxjs';

const observable = new Observable(function subscribe(observer) {
  const id = setInterval(() => {
    observer.next('hi')
  }, 1000);
});
```

A função de subscrição é que define o comportamento!
É executada quando se chama o método *subscribe()* sobre o objeto Observable

RxJS

- Exemplos: inscrevendo a um Observable

```
observable.subscribe(x => console.log(x));
```



Essa função representa um objeto
Observer

RxJS

- Exemplos: criação de um Observable

```
import { Observable } from 'rxjs';

const observable = Observable.create(function subscribe(observer) {
  try {
    observer.next(1);
    observer.next(2);
    observer.next(3);
    observer.complete();
  } catch (err) {
    observer.error(err);
  }
});
```

RxJS

- Exemplos: inscrevendo-se em um Observable

```
observable.subscribe(  
  x => console.log('Observer got a next value: ' + x),  
  err => console.error('Observer got an error: ' + err),  
  () => console.log('Observer got a complete notification')  
);
```

```
const observer = {  
  next: x => console.log('Observer got a next value: ' + x),  
  error: err => console.error('Observer got an error: ' + err),  
  complete: () => console.log('Observer got a complete notification'),  
};  
  
observable.subscribe(observer);
```

RxJS e Angular

- Objetos *Observable* são utilizados pelo Angular em diversos momentos tais como requisições HTTP, pipe async, roteamento, formulários e tratamento de eventos
- Importante: nunca esqueça de cancelar a subscrição no objeto Observable
 - Evita problemas de vazamento de memória e comportamento indesejado
 - Utilize o evento de ciclo de vida `ngOnDestroy()` para efetuar a tarefa

```
subscription: Subscription;

ngOnInit() {
  this.subscription = umObservable.subscribe(...);
}

ngOnDestroy() {
  this.subscription.unsubscribe();
}
```

Angular e HTTP



Angular e HTTP

- Angular fornece a classe *HttpClient* no módulo Angular *HttpClientModule* no módulo JavaScript `@angular/common/http` para implementar o acesso a recursos via o protocolo HTTP
 - Importante: a instância da classe *HttpClient* deve ser obtida via injeção de dependências
- Documentação:
 - <https://angular.io/guide/http>
 - <https://angular.io/api/common/http/HttpClient>

Angular e HTTP

- Exemplos: configuração e injeção

```
import { HttpClientModule } from '@angular/common/http';  
...  
@NgModule({  
  imports: [ BrowserModule, HttpClientModule ],  
  declarations: [ AppComponent ],  
  bootstrap: [ AppComponent ]  
})  
export class AppModule {}
```

```
import { Injectable } from '@angular/core';  
import { HttpClient } from '@angular/common/http';  
@Injectable()  
export class ConfigService {  
  constructor(private http: HttpClient) { }  
}
```


Angular e HTTP

- Os métodos de *HttpClient* retornam objetos *Observable*
 - A resposta de uma requisição está encapsulada no objeto *Observable*
 - O tipo de *Observable* utilizado somente emite um único valor e depois completa
- **IMPORTANTE:**
 - *HttpClient* não inicia uma requisição até que seja chamado o método *subscribe()* sobre o objeto *Observable* retornado

Angular e HTTP

- Método `get()`
 - Método genérico de requisição GET
 - Dois parâmetros: URL, opções para configuração da requisição
 - Se um tipo genérico for utilizado, o corpo da resposta é um objeto tipado
 - Se um tipo genérico não for utilizado, o corpo da resposta é um objeto JSON sem tipo associado
 - Propriedade *observe* do objeto de configuração permite acessar a informação desejada (*body*, *response*); *body* é o padrão; *response* retorna um objeto *HttpResponse* com toda a resposta
 - Usualmente é necessário utilizar o operador *map()* do RxJS para processar objetos complexos no corpo da resposta

Angular e HTTP

- Exemplos: get

```
getHeroes (): Observable<Hero[]> {  
    return this.http.get<Hero[]>(this.heroesUrl)  
}
```

```
getConfigResponse(): Observable<HttpResponse<Config>> {  
    return this.http.get<Config>(this.configUrl, { observe: 'response' });  
}
```

Angular e HTTP

- Tratamento de erros
 - Requisições HTTP podem resultar em erros
 - RxJS fornece o operador *catchError()* para ser utilizado via método *pipe()* de Observable
 - Método *pipe()* permite encadear diversas operações
 - *catchError()* intercepta um Observable que falhou e passa um objeto *HttpErrorResponse* para o tratador
 - Dois tipos de erros suportados:
 - Erros de resposta – respostas HTTP de falha, tais como 400 ou 500
 - Exceções – respostas que produzem objetos *ErrorEvent*
 - *retry()* realiza um determinado número de tentativas se a requisição falhou

Angular e HTTP

- Exemplos: get com tratamento de erro

```
getHeroes (): Observable<Hero[]> {  
  return this.http.get<Hero[]>(this.heroesUrl)  
    .pipe(  
      retry(3),  
      catchError(this.handleError('getHeroes', []))  
    );  
}
```

Angular e HTTP

- Exemplos: get com tratamento de erro

```
private handleError<T> (operation = 'operation', result?: T) {  
  return (error: HttpResponse): Observable<T> => {  
    if (error.error instanceof ErrorEvent) {  
      console.error('Erro:', error.error.message);  
    } else {  
      console.error('Status:', error.status);  
    }  
    return of(result as T); // return Observable com resultado  
    OU  
    return throwError('Falha na requisição'); // return ErrorObservable  
  };  
}
```

Angular e HTTP

- Método put()
 - Método requisição PUT
 - Três parâmetros: URL, dados, opções para configuração da requisição
- Método post()
 - Método requisição POST
 - Três parâmetros: URL, dados, opções para configuração da requisição
- Método delete()
 - Método genérico de requisição DELETE
 - Dois parâmetros: URL, opções para configuração da requisição

Angular e HTTP

- Exemplos: put

```
const httpOptions = {  
  headers: new HttpHeaders({ 'Content-Type': 'application/json' })  
};  
  
updateHero (hero: Hero): Observable<any> {  
  return this.http.put(this.heroesUrl, hero, httpOptions).pipe(  
    catchError(this.handleError<any>('updateHero'))  
  );  
}
```


Angular e HTTP

- Exemplos: post

```
const httpOptions = {  
  headers: new HttpHeaders({ 'Content-Type': 'application/json' })  
};  
  
addHero (hero: Hero): Observable<Hero> {  
  return this.http.post<Hero>(this.heroesUrl, hero, httpOptions).pipe(  
    catchError(this.handleError<Hero>('addHero'))  
  );  
}
```

Angular e HTTP

- Exemplos: delete

```
deleteHero (id: number): Observable<Hero> {  
  const url = `${this.heroesUrl}/${id}`;  
  return this.http.delete<Hero>(url, httpOptions).pipe(  
    catchError(this.handleError<Hero>('deleteHero'))  
  );  
}
```