

Angular

Serviços

Ely – elydasilvamiranda@gmail.com

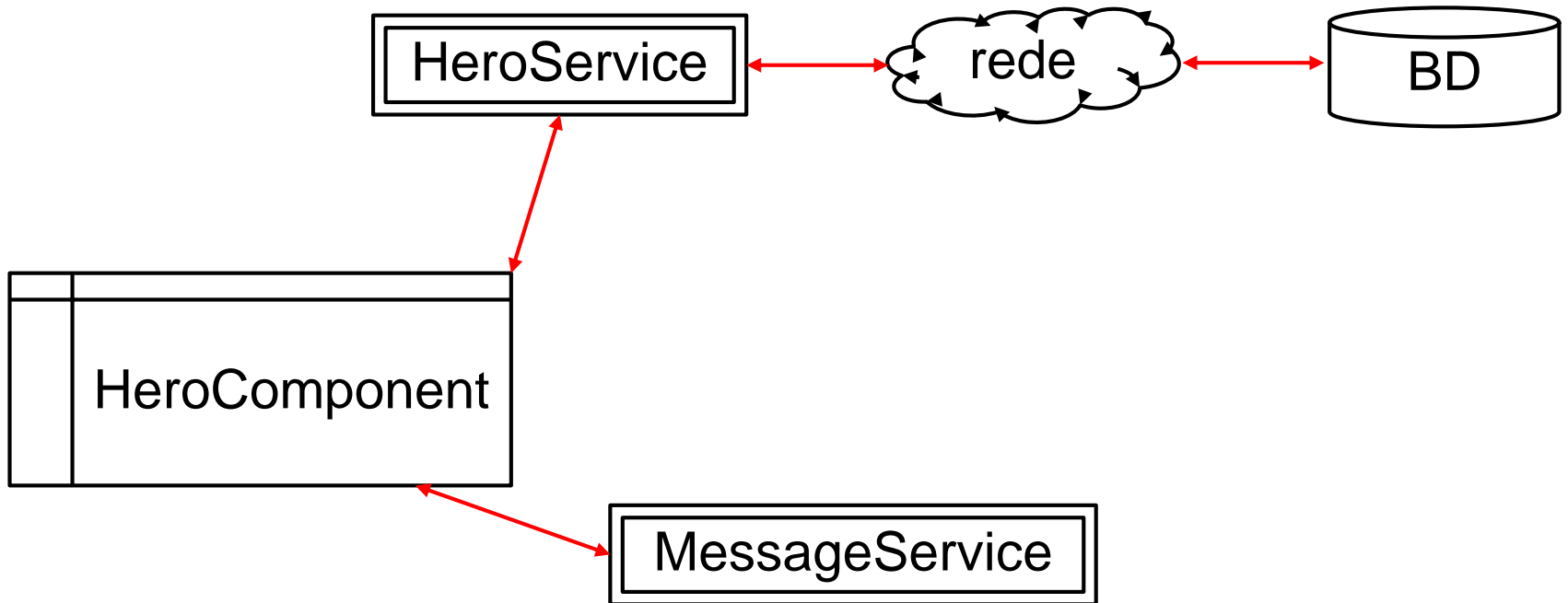
Serviços

- O componente Heroes está atualmente recebendo e exibindo dados falsos;
- Após uma refatoração, adicionaremos dados vindos de um servidor WEB para a aplicação.

Serviços

- Os componentes não devem buscar ou salvar dados diretamente;
- Eles devem se concentrar em apresentar dados e delegar acesso a dados a uma classe específica;
- Assim, uma recomendação é sempre isolar as requisições à APIs em classes específicas;
- Essas classes são conhecidas como classes de serviço (services).

Serviços



Serviços

- Algumas convenções:
 - O nome do arquivo deve usar o padrão `<nome_componente>.service.ts`;
 - O arquivo deve ficar na mesma pasta da classe de modelo a que ele acessa;
- Criaremos dois serviços:
 - HeroService: usado para obter heróis e usado via injeção de dependência;
 - MessageService: usado para compartilhar feedbacks ao usuário pela aplicação.

Criando um serviço

- Criaremos um HeroService para encapsular as requisições à uma futura API;
- Ainda não acessaremos uma API, mas vamos transferir os dados “mockados” para o serviço;
- Para criar o serviço via CLI, digite:
 >> ng generate service hero
- O comando gera a classe HeroService em src/app/hero.service.ts.

HeroService

- Arquivo hero.service.ts:

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

@Injectable({
  providedIn: 'root'
})
export class HeroService {

  constructor() { }
}
```

Injeção de dependência

- O serviço criado importa e utiliza o decorador `@Injectable()`;
- Isso informa que classe de serviço pode ser injetada em outras classes:
 - Ao usar o serviço, não será necessário “instanciar” usando um operador `new`;
 - Evita que objetos de serviços sejam criados indistintamente;
 - Também permite ao Angular otimizar o aplicativo removendo-o caso ele não seja usado.

Obtendo os dados dos Heróis

- Para tornar o serviço utilizável, devem ser importadas:
 - A classe Hero;
 - Os dados do arquivo mock-heroes.ts;
- Além disso, implementar um método “getHero()” que retorna os heróis.

Obtendo os dados dos Heróis

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

```
import { Hero } from '../heroes/hero';
```

```
import { HEROES } from '../mock-heroes';
```

```
@Injectable({  
  providedIn: 'root'
```

```
})
```

```
export class HeroService {
```

```
  constructor() { }
```

```
  getHeroes(): Hero[] {
```

```
    return HEROES;
```

```
  }
```

```
}
```

Disponibilizando o HeroService

- Você deve disponibilizar o HeroService para o sistema de injeção de dependência;
- Com isso, não será necessário instanciar o serviço diretamente;
- Fazemos isso registrando um provedor:
 - Um provedor é algo que pode criar ou entregar um serviço e gerenciará classe HeroService;
 - O provedor *root* é o objeto responsável por escolher e injetar o serviço onde é necessário.

Disponibilizando o HeroService

- Por padrão, o CLI faz isso no decorador provedor no decorador @Injectable:

```
@Injectable({  
    providedIn: 'root'  
})
```

- Quando se registra um serviço no nível raiz:
 - O Angular cria um Singleton (única instância compartilhada);
 - Injeta em qualquer classe que o solicite;

Usando o serviço

- Para usar o serviço em HeroComponent, é preciso :
 - Importar o serviço e criar um atributo no construtor;
 - Substituir a declaração da variável heroes por uma coleção vazia:

```
import { Component, OnInit } from '@angular/core';
import { Hero } from '../hero';
import { HeroService } from '../hero.service';
//..
export class HeroesComponent implements OnInit {
  heroes: Hero[];
  selectedHero: Hero;
  constructor(private heroService: HeroService) {}
  //..
}
```

Usando o serviço

- Para usar o serviço em HeroComponent, é preciso :
 - Criar um método que usa o serviço;
 - Chamar o método acima no ngOnInit():

```
//..  
export class HeroesComponent implements OnInit {  
  heroes: Hero[];  
  //  
  getHeroes(): void {  
    this.heroes = this.heroService.getHeroes();  
  }  
  
  ngOnInit() {  
    this.getHeroes();  
  }  
  //  
}
```

Sobre o método `ngOnInit()`

- Embora seja possível chamar `getHeroes()` no construtor, essa não é a melhor prática;
- Devemos deixar o construtor para uma inicializações simples:
 - Criar propriedades e inicializar atributos;
 - O construtor não deve fazer nada, muito menos executar regras de negócio ou requisições HTTP.

Sobre o método `ngOnInit()`

- Em vez disso, o `getHeroes()` deve ser chamado dentro `ngOnInit`;
- Pelo ciclo de vida do Angular, é chamado em um momento apropriado após o construtor.

Métodos síncronos

- O método `HeroService.getHeroes ()` tem uma assinatura síncrona:

```
this.heroes = this.heroService.getHeroes();
```

- Até o momento, isso funciona na aplicação, pois os dados são fictícios e estão na máquina local;
- Porém, uma chamada síncrona bloqueia a interface até que ela seja finalizada;
- Para grandes volumes de dados, via rede, isso pode ser um problema.

Observable

- O ideal é trabalhar de forma assíncrona e o `getHeroes()` ter uma assinatura assíncrona;
- Uma forma de fazer chamadas assíncronas com o Angular é usar classes Observable;
- Observables são itens fundamentais utilizados com programação reativa.

Observable

- Um observable nos dá operadores para:
 - Cancelar *requests* para poupar processamento;
 - Fazer uma nova requisição caso algum problema na conexão aconteça.

<https://angular.io/guide/observables>

<https://medium.com/tableless/entendendo-rxjs-observable-com-angular-6f607a9a6a00>

Observable

- Para alterar a classe HeroService para utilizar o padrão, devemos:
 - Adicionar à assinatura do método getHeroes().
 - Utilizar a função of(HEROES) que retorna um Observable<Hero[]> ;

```
// ...
```

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

```
// ...
```

```
export class HeroService {
```

```
  // ...
```

```
  getHeroes(): Observable<Hero[]> {
```

```
    return of(HEROES);
```

```
  }
```

```
}
```

Consumindo um observable

- Um dos operadores mais importantes do *Observable* é o *subscribe*:
- Assim, a aplicação espera de forma não bloqueante a resposta;
- Métodos que consomem um observable fazem então uma “subscription”;
- Assim, o HeroComponent terá que fazer uma subscrição do observable retornado pelo serviço.

Consumindo um observable

- Componente HeroesComponent:

```
//..  
export class HeroesComponent implements OnInit {  
    heroes: Hero[];  
  
    //..  
    getHeroes(): void {  
        this.heroService.getHeroes()  
            .subscribe(heroes => this.heroes = heroes);  
    }  
    //..  
}
```

Consumindo um observable

- O método `Observable.subscribe()` é a diferença crítica:
 - A versão anterior do `getHeroes()` ocorre de forma síncrona;
 - Isso não funcionaria de forma instantânea quando o serviço estivesse fazendo solicitações remotas;
- A nova versão espera que o `Observable` finalize a requisição, entretanto não bloqueia a interface.

Componente de mensagens

- Criaremos um componente e um serviço para trabalhar com feedbacks ao usuário na aplicação:
 - MessagesComponent que exibirá mensagens do aplicativo na parte inferior da tela;
 - MessageService: serviço injetável para enviar mensagens a serem exibidas;
 - Injetaremos o MessageService no HeroService;
 - Exibiremos uma mensagem quando HeroService busca heróis com sucesso.

Componente de mensagens

- Criaremos um componente e um serviço para trabalhar com feedbacks ao usuário na aplicação:

18	Dr IQ
19	Magma
20	Tornado

Messages

clear

HeroService: fetched heroes

Criando o componente

- Via CLI:
 >> `ng generate component messages`
- Os arquivos de componentes serão criados em na pasta `src/app/messages`;
- Além disso, o CLI declara o `MessagesComponent` no `AppModule`.

Alterando a visão

- Em app.componente.html devemos adicionar o componente ao final:

```
<h1>Welcome to {{ title }}!</h1>
```

```
<app-heroes></app-heroes>
```

```
<app-messages></app-messages>
```

Criando um serviço

- Via CLI:
 >> ng generate service message

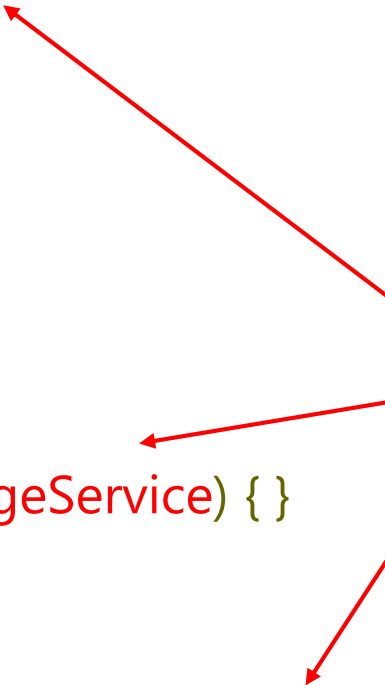
Editando o serviço

- Em message.service.ts:

```
import { Injectable } from '@angular/core';
@Injectable({
  providedIn: 'root',
})
export class MessageService {
  messages: string[] = [];
  add(message: string) {
    this.messages.push(message);
  }
  clear() {
    this.messages = [];
  }
}
```

MessageService no HeroesService

```
//..  
import { MessageService } from '../message.service';  
  
@Injectable({  
  providedIn: 'root',  
})  
export class HeroService {  
  
  constructor(private messageService: MessageService) { }  
  
  getHeroes(): Observable<Hero[]> {  
    this.messageService.add('HeroService: fetched heroes');  
    return of(HEROES);  
  }  
}
```



MessagesComponent

```
import { Component, OnInit } from '@angular/core';
import { MessageService } from '../message.service';

@Component({
  selector: 'app-messages',
  templateUrl: './messages.component.html',
  styleUrls: ['./messages.component.css']
})

export class MessagesComponent implements OnInit {

  constructor(public messageService: MessageService) {}

  ngOnInit() {
  }

}
```



messages.component.html

```
<div *ngIf="messageService.messages.length">

  <h2>Messages</h2>

  <button class="clear"
          (click)="messageService.clear()">clear
</button>

  <div *ngFor="let message of messageService.messages">
    {{message}}
  </div>

</div>
```


messages.component.css

```
h2 {
  color: red;
  font-family: Arial, Helvetica,
               sans-serif;
  font-weight: lighter;
}
body { margin: 2em; }
body, input[text], button {
  color: crimson;
  font-family: Cambria, Georgia;
}
button.clear {
  font-family: Arial;
  background-color: #eee;
  border: none;
  padding: 5px 10px;
  border-radius: 4px;
  cursor: pointer;
  cursor: hand;
}

button:hover {
  background-color: #cfd8dc;
}
button:disabled {
  background-color: #eee;
  color: #aaa;
  cursor: auto;
}
button.clear {
  color: #333;
  margin-bottom: 12px;
}
```

Angular

Serviços

Ely – elydasilvamiranda@gmail.com