08

Isolando a acesso em serviços

Transcrição

Para melhorarmos o código, primeiro voltaremos a `app.component.ts` para corrigir a URL que alteramos anteriormente (`http://localhost:3000/flavio/photos`), e então salvaremos e iremos ao navegador para conferir que tudo funciona conforme esperado. O problema desta abordagem é que, supondo que temos muito mais componentes, e que em sete deles precisamos dar acesso à API de imagens, teremos o seguinte trecho se repetindo em sete locais diferentes:

```
http
    .get<Object[]>('http://localhost:3000/flavio/photos')
    .subscribe(photos => this.photos = photos);
```

COPIAR CÓDIGO

Além disso, se o endereço da API for alterado, teremos que lembrar de modificá-lo em todos os outros lugares em que aparece. O componente não precisa saber de detalhes de onde os dados estão vindo. Portanto, isolaremos o acesso à API em uma classe especializada em consumi-las.

No universo do Angular, chamamos estas classes de **serviços**. Na pasta "photo", ou seja, na mesma pasta de `photo.component.html` e `photo.component.ts`, criaremos o arquivo `photo.service.ts`, por motivos de organização e praticidade. Este novo arquivo não é um componente, no entanto continuará sendo uma classe.

Para poder acessar a API, o `PhotoService` também depende de `HttpClient`, então acrescentaremos `constructor` e a propriedade `http`, tomando cuidado para fazermos a importação de `angular/common/http`. Ele terá o método `listFromUser()`, que receberá como parâmetro um `userName`.

Deste modo, quem utilizar o `PhotoService` chamará este método passando `userName`, no caso, trabalharemos com `flavio`, e retornará todos os dados daquela API.

```
import { HttpClient } from "@angular/common/http";

export class PhotoService {

        constructor(http: HttpClient) {}

        listFromUser(userName) {

        }
}
```

COPIAR CÓDIGO

Feito isso, voltaremos a `app.component.ts`, copiaremos o código referente a `http`, e o colaremos em `photo.service.ts`:

```
import { HttpClient } from "@angular/common/http";

export class PhotoService {

        constructor(http: HttpClient) {}

        listFromUser(userName) {

            http
            .get<Object[]>('http://localhost:3000/flavio/pho
            .subscribe(photos => this.photos = photos);
```

```
        }
}
```

COPIAR CÓDIGO

No entanto, não temos acesso ao `HttpClient` no método `listFromUser()`, por isso teríamos que incluir algo como `http: HttpClient` e, no construtor, `this.http = http`, já que ele, sim, tem acesso. No método, usaríamos `this.http`, e ficaríamos com o código desta forma:

```
export class PhotoService {

        http: HttpClient;

        constructor(http: HttpClient) {

            this.http = http;

        }

        listFromUser(userName) {

            this.http
            .get<Object[]>('http://localhost:3000/flavio/photo
            .subscribe(photos => this.photos = photos);

        }
}
```

COPIAR CÓDIGO

Porém, não é isso que faremos. Vamos utilizar o modificador de acesso `private` no parâmetro do construtor. Assim, o TypeScript entende que queremos não apenas receber este parâmetro como torná-lo uma propriedade da classe. Com o `private`, tudo que estiver fora de `PhotoService` não poderá

usar o `http`, e isto justifica o uso de `this.http` também. Sem o modificador, teremos um erro de compilação.

Também não podemos usar o `subscribe()` aqui, já que isto deverá ser feito no momento da busca de dados. O responsável por isso é aquele que for utilizar o método `listFromUser()`. Lembrando que, da maneira em que está, o `observable`, por ser *lazy*, não buscará os dados.

Há mais um detalhe: qual o tipo do `userName`? É `any`, o que quer dizer que qualquer tipo que passarmos será aceito. No entanto, sabemos que `userName` será sempre texto, então podemos indicar isto. E se passarmos o mouse sobre `this`, será exibido que o retorno é `void`, que equivale a nada, sendo assim acrescentaremos `return`:

```
listFromUser(userName: string) {

    return this.http
    .get<Object[]>(API + '/flavio/photos')

}
```

COPIAR CÓDIGO

Precisamos tomar cuidado para não usarmos **S**tring, com a primeira letra em maiúsculo.

Deste modo, ao passarmos o mouse em `this`, será retornado um *Observable* do tipo `Object[]`. Salvaremos o arquivo, abriremos `app.component.ts`, deletaremos a importação de `HttpClient` e importaremos `photoService` do tipo `PhotoService`, tornando o código muito mais limpo:

```
export class AppComponent {

    photos: Object[] = [];
```

```
    constructor(photoService: PhotoService) {

        photoService
            .listFromUser('flavio')
            .subscribe(photos => this.photos = photos);
    }
}
```

COPIAR CÓDIGO

Voltaremos ao navegador após salvarmos todas as alterações, e haverá um erro de ausência de provedor para `PhotoService`. Isto ocorre porque quando o Angular cria `AppComponent` e tenta injetar `photoService`, não consegue, por desconhecê-lo. É um pouco diferente do que vimos anteriormente: ao criarmos um serviço, usamos `@Injectable()`.

Este *decorator*, que é automaticamente importado do `angular/core`, indica que `photoService` é injetável, ou seja, pode receber `HttpClient` e outros. No entanto, precisamos informar o seu escopo, se será um único `PhotoService` para a aplicação inteira, ou não.

No nosso caso, se tivermos trinta componentes e quisermos usar o `PhotoService`, e o mesmo objeto, então passaremos a configuração `providedIn`, um objeto JavaScript cujo valor é `root`. Com isso, sinalizamos que quando o Angular for criá-lo, será no **escopo raiz**, isto é, qualquer componente da nossa aplicação que precisar de `PhotoService` o terá disponível.

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

    constructor(private http: HttpClient) {}

    listFromUser(userName: string) {
        return this.http
```

```
            .get<Object[]>(API + '/' + userName + '/photos');
    }
}
```

COPIAR CÓDIGO

Salvaremos, retornaremos ao navegador, e tudo continuará funcionando muito bem. E nosso código está organizado de maneira muito mais elegante, inclusive em termos de manutenção e legibilidade. Todavia podemos melhorá-lo ainda mais, e veremos como fazer isso em breve.