



APLICAÇÕES INTERATIVAS

SERVIÇOS

Componentes e serviços

- Não devemos utilizar componentes para o gerenciamento de dados
- Componentes devem se concentrar em apresentar os dados visualmente e lidar com eventos da interface

Serviços

- Os serviços em angular são normalmente utilizados para gerenciamento de dados, embora tenham propósito geral
- Enquanto os componentes fazem a interface com o usuário, os serviços fazem a interface com os dados

Serviços

- Os componentes e serviços conversarão de forma que um lide com a interface e os eventos do usuário e, o outro, com a busca e salvamento de informações

Serviços

- Também é possível concentrar funções comumente utilizadas e outros códigos que possam ser reutilizados entre componentes utilizando serviços

CRUD

- Acrônimo para “Create, Read, Update, Delete” ou “Criar, Ler, Atualizar, Deletar”
- Normalmente refere-se a funcionalidade de cadastro (com todas as operações básicas padrão)

Serviços de CRUD

- É muito comum os serviços serem utilizados para gerenciamento de um CRUD de um item de modelo
- O serviço deverá ter uma função para cada operação do CRUD

Serviços de CRUD

- As funções receberão e retornarão o item de modelo conforme necessário, obtendo-o e persistindo-o numa fonte de dados
- A fonte de dados pode ser qualquer local, como um serviço remoto, o *local storage* do navegador ou mesmo a memória

Serviços de CRUD

- O importante é que os demais elementos da aplicação NÃO devem saber qual a fonte de dados, isso é responsabilidade do serviço
- O que devem saber é que, para persistir ou obter elementos de negócio, devem chamar o respectivo serviço

Trabalhando com Serviços

- Vamos criar uma aplicação de gerenciamento de jogos
- A aplicação armazenará dados em memória através de um serviço

Trabalhando com Serviços

- Posteriormente, converteremos o serviço para lidar com local storage
- O restante da aplicação não precisará de mudanças, pois todo o gerenciamento dos dados com relação a sua origem estará concentrado no serviço

Nova Aplicação

- Construiremos a aplicação começando pelo modelo, serviço e, em seguida, componente visual

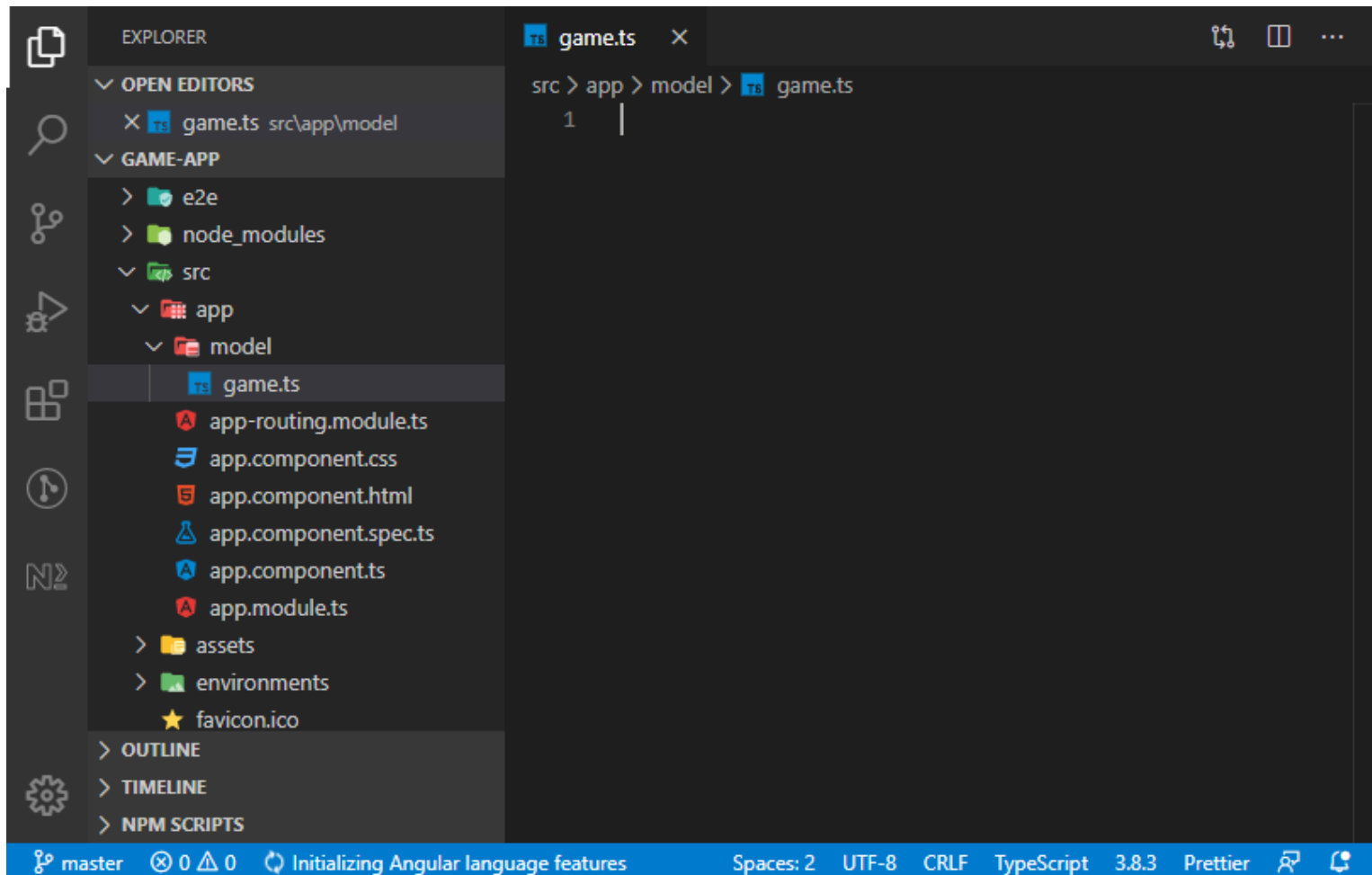
Nova Aplicação

- Crie um novo projeto
- Chame-o de “game-app”
- Abra a pasta do projeto no VS Code

Classe de Modelo

- Crie um pacote na pasta “app” chamado “model”
- Em seu interior, crie um arquivo chamado “game.ts”

Classe de Modelo



Classe de Modelo

- Defina a classe de modelo “game”:

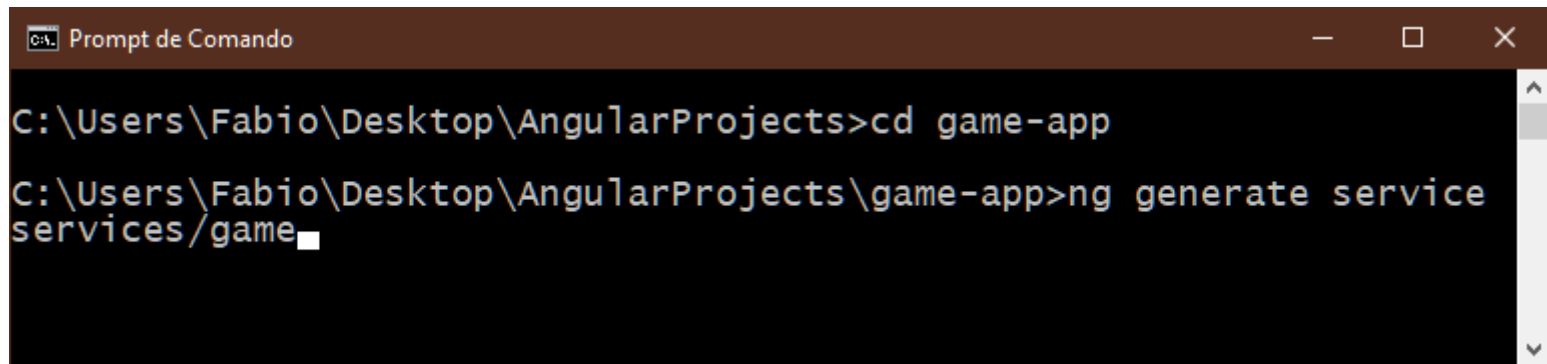
```
export class Game {  
  id?: number;  
  name: string;  
  genre: string;  
  platform: string;  
  status = 'na';  
  
  constructor(name: string,  
              genre: string, platform: string) {  
    this.name = name;  
    this.genre = genre;  
    this.platform = platform;  
  }  
}
```

Criando um serviço

- Para criar um serviço, também utilizamos o ***ng generate, junto a service*** e o **nome do serviço a gerar** (geralmente relacionado a um item de modelo)
- Também podemos especificar uma pasta para o serviço

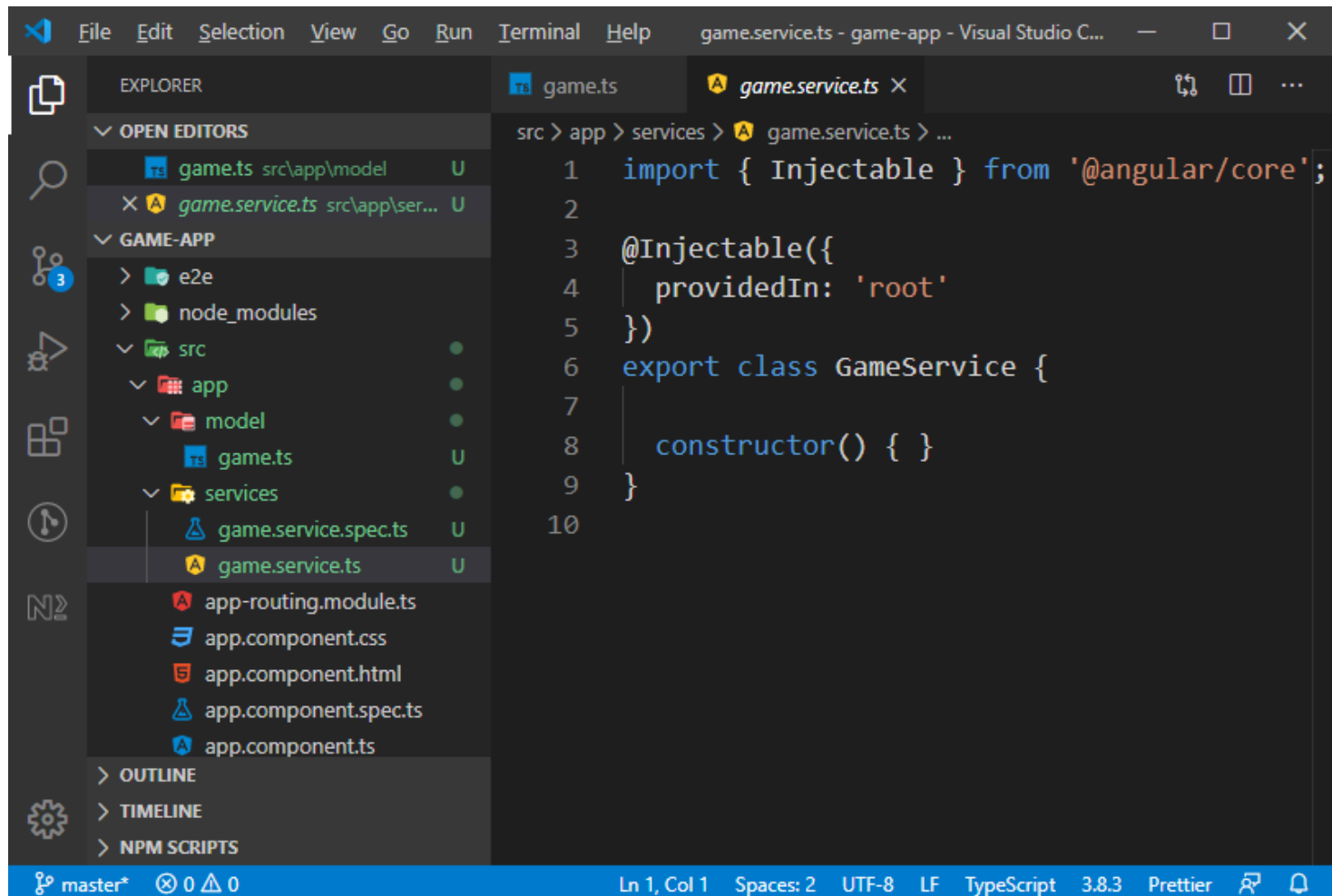
Criando um serviço

- Para criar um serviço de jogos, em um prompt de comando e na pasta do projeto, digite:
- `ng generate service services/game`



```
C:\Users\Fabio\Desktop\AngularProjects>cd game-app  
C:\Users\Fabio\Desktop\AngularProjects\game-app>ng generate service  
services/game
```

Serviço criado



The screenshot shows the Visual Studio Code interface with the Explorer sidebar on the left and the Editor on the right. The Explorer sidebar shows the project structure for 'GAME-APP', with the 'services' folder expanded. The 'game.service.ts' file is selected. The Editor shows the code for 'game.service.ts', which includes an import from '@angular/core', an @Injectable decorator, and a class definition for 'GameService'.

```
src > app > services > game.service.ts > ...
1  import { Injectable } from '@angular/core';
2
3  @Injectable({
4    providedIn: 'root'
5  })
6  export class GameService {
7
8    constructor() { }
9  }
10
```

Visual Studio Code status bar at the bottom: master* 0 0 0 Ln 1, Col 1 Spaces: 2 UTF-8 LF TypeScript 3.8.3 Prettier

Serviço de jogos

- O serviço de jogos deverá permitir executar as principais operações de CRUD, a saber:
 - Inserir jogo
 - Excluir jogo
 - Editar um jogo
 - Listar os jogos salvos

Serviço de jogos

- Inicialmente, nosso serviço salvará todos os dados em memória
- **Para isso, definimos um atributo de array do elemento de negócio que desejamos gerenciar**
- **O array será o espaço na memória para armazenar os jogos salvos**

Serviço de jogos

- Como só há uma instância de cada serviço na memória por vez no Angular, nossos dados ficarão salvos enquanto a aplicação estiver carregada e aberta
- **Uma vez descarregada (ou recarregada) os dados serão perdidos**

Serviço de jogos

- Em breve, vamos atualizar o serviço de forma que o mesmo utilize o local storage para armazenar os dados
- **Desta forma, os dados ficarão gravados de forma permanente**

Array de dados no Serviço

- Vamos adicionar o array de jogos como uma propriedade no serviço
- **O array será privado. Só o serviço poderá modifica-lo**
- **Métodos serão criados para permitir que o resto da aplicação interaja com os dados**

Array de dados no Serviço

- Isso é feito visando proteger a origem dos dados da aplicação
- **Se a aplicação não tem acesso direto a fonte de dados, esta funcionalidade fica encapsulada no serviço e permite alterá-la e ajustá-la facilmente, sem afetar o resto do app**

Array de dados no Serviço

- Adicione a propriedade a seguir no serviço:

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

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

```
export class GameService {  
  private games = new Array<Game>();
```

```
  constructor() { }  
}
```

Método de inserção

- Crie o método de inserção, que permita adicionar itens ao array
- **O método deverá receber o jogo a inserir e adicioná-lo no array utilizando o método push de arrays (adiciona um item ao último espaço disponível no array)**

Método de inserção

- Precisamos que a inserção defina um id para cada novo elemento inserido
- **Desta forma, será possível editar e remover os itens pelo ID apenas e o resto da aplicação não precisa se preocupar com a definição de um ID**

Método de inserção

- Para isso, criaremos também um campo numérico iniciado em 0 e capaz de ser incrementado a cada inserção
- **Este campo será usado como referência de ID**

Método de inserção

```
import { Injectable } from "@angular/core";
import { Game } from "../model/game";

@Injectable({
  providedIn: "root",
})
export class GameService {
  private games = new Array<Game>();
  private autoGeneratedId = 0;

  constructor() {}

  insert(game: Game): void {
    game.id = this.autoGeneratedId;
    this.games.push(game);
    this.autoGeneratedId++;
  }
}
```

Método de listagem

- O método de listagem do serviço do CRUD deve retornar todos os elementos salvos disponíveis no cadastro
- No nosso caso, já que todos os jogos salvos em memória estão no array, basta retorná-lo

Método de listagem

...

```
insert(game: Game) {  
    game.id = this.autoGeneratedId;  
    this.games.push(game);  
    this.autoGeneratedId++;  
}
```

```
list(): Array<Game> {  
    return this.games;  
}  
}
```

Método de exclusão

- O método de exclusão deverá localizar o item no array e removê-lo
- Para tanto, é possível utilizar a função *splice* do array
- *Splice* permite remover um ou mais itens em determinada posição do array

Método de exclusão

- Desta forma, vamos usar um for para varrer o array até encontrar o item que tem o ID do item que se deseja remover
- Quando isso acontecer, o método **splice** do array será acionado, informando a posição encontrada para remover o elemento do array

Método de exclusão

```
...  
list(): Array<Game> {  
    return this.games;  
}  
  
remove(id: number): void {  
    for (let i = 0; i < this.games.length; i++) {  
        const g = this.games[i];  
        if (g.id === id) {  
            this.games.splice(i, 1);  
            break;  
        }  
    }  
}  
...  
...
```

Método de exclusão alternativo

- Também é possível utilizar o método de filtragem de itens do array para uma estrutura mais compacta

Método de exclusão alternativo

...

```
remove(id: number): void {  
    this.games = this.games.filter((game) => {  
        return game.id !== id;  
    });  
}
```

...

Método de edição

- O método de edição deve permitir informar os novos dados de um item, mantendo o ID
- **Quando o item com o ID respectivo for encontrado, terá seus dados sobrescritos com os novos dados fornecidos**

Método de edição

- No nosso caso, é possível que o método de edição receba o item de jogo já preenchido com os novos dados, mas mantendo o ID antigo
- Quando o item com ID for encontrado, será substituído pelo novo item

Método de edição

- Desta forma, semelhante ao método de remoção, vamos percorrer o array. No entanto, ao invés de remover o item com o splice, quando o item com o ID correspondente for encontrado, será substituído pelo jogo fornecido como parâmetro

Método de edição

```
...  
  update(game: Game): void {  
    for (let i = 0; i < this.games.length; i++) {  
      const g = this.games[i];  
      if (g.id === game.id) {  
        this.games[i] = game;  
        break;  
      }  
    }  
  }  
}  
...  
...
```

Método de edição alternativo

- Alternativamente, é possível usar a função `findIndex` do Array para encontrar o índice do item desejado e atribuir o item atualizado na posição encontrada

Método de edição alternativo

```
...  
  update(game: Game): void {  
    const i = this.games.findIndex(g => g.id === game.id);  
    if (i >= 0) {  
      this.games[i] = game;  
    }  
  }  
...  

```

Implementação do Serviço

- Desta forma, o serviço está completo e poderá ser utilizado em qualquer componente visual que necessitar interagir com dados de jogos

Restante de Aplicação

- Vamos seguir criando os demais elementos da aplicação, principalmente os elementos do componente visual, que irão interagir com o serviço para possibilitar que o usuário seja capaz de manipular os dados

Componente Visual

- Crie um novo componente visual, responsável pelo gerenciamento de jogos
- Chame-o de “game”

Componente Visual

```
C:\Users\Fabio\Desktop\AngularProjects\game-app>ng generate component
views/game
CREATE src/app/views/game/game.component.html (19 bytes)
CREATE src/app/views/game/game.component.spec.ts (614 bytes)
CREATE src/app/views/game/game.component.ts (267 bytes)
CREATE src/app/views/game/game.component.css (0 bytes)
UPDATE src/app/app.module.ts (473 bytes)

C:\Users\Fabio\Desktop\AngularProjects\game-app>
```


Importando e injetando o serviço

- Será necessário injetar o serviço no componente para que o mesmo possa utilizar o serviço em suas operações
- Para tanto, é preciso importar o serviço no componente e, em seguida, adicionar o código de injeção

```
import { Component, OnInit } from "@angular/core";
import { GameService } from 'src/app/services/game.service';

@Component({
  selector: "app-game",
  templateUrl: "../game.component.html",
  styleUrls: ["../game.component.css"],
})
export class GameComponent implements OnInit {

  constructor(private gameService: GameService) {}

  ...
}
```

Vetor de Itens de Modelo

- Defina um vetor de itens de modelo na classe *“game.component.ts”*
- Este será o vetor que armazenará os itens para uso e manipulação temporária, na tela, e sempre será atualizado com o vetor do serviço

Vetor de Itens de Modelo

```
import { Component, OnInit } from "@angular/core";  
import { Game } from "src/app/model/game";  
import { GameService } from 'src/app/services/game.service';
```

```
@Component({  
  selector: "app-game",  
  templateUrl: "./game.component.html",  
  styleUrls: ["./game.component.css"],  
})  
export class GameComponent implements OnInit {  
  games = new Array<Game>();  
  
  constructor(private gameService: GameService) {}  
  
  ...
```

Itens do Array

- Como vimos anteriormente, para utilizar uma classe de modelo, é necessário instanciá-la (obter um objeto que possa armazenar valores)
- Cada objeto instanciado é único e independente dos demais

Itens do Array

- Como o vetor é um vetor de objetos da classe game, aceitará objetos desse tipo
- Para listarmos todos os jogos armazenados, basta utilizar um `*ngFor` na página, iterando pelo array

Listando Itens

```
<h2>Meus Jogos</h2>
<ul>
  <li *ngFor="let game of games">
    {{game.name}} - {{game.status}}
  </li>
</ul>
```

Listagem

- Para que a lista do componente esteja sempre atualizada com a lista principal no serviço, vamos criar um método que, quando chamado, será responsável por atualizar os valores do componente com os do serviço

Listagem

- Esse método será chamado toda vez que uma operação de alteração (inserção, edição ou deleção) for efetuada, para manter a lista do componente atualizada
- Também será chamado assim que o componente carregar, para que os dados já apareçam na tela (se houver)

Listagem

- Crie o método de atualização da lista de jogos conforme a seguir

...

```
refreshGames(): void {  
    this.games = this.gameService.list();  
}
```

...

Listagem

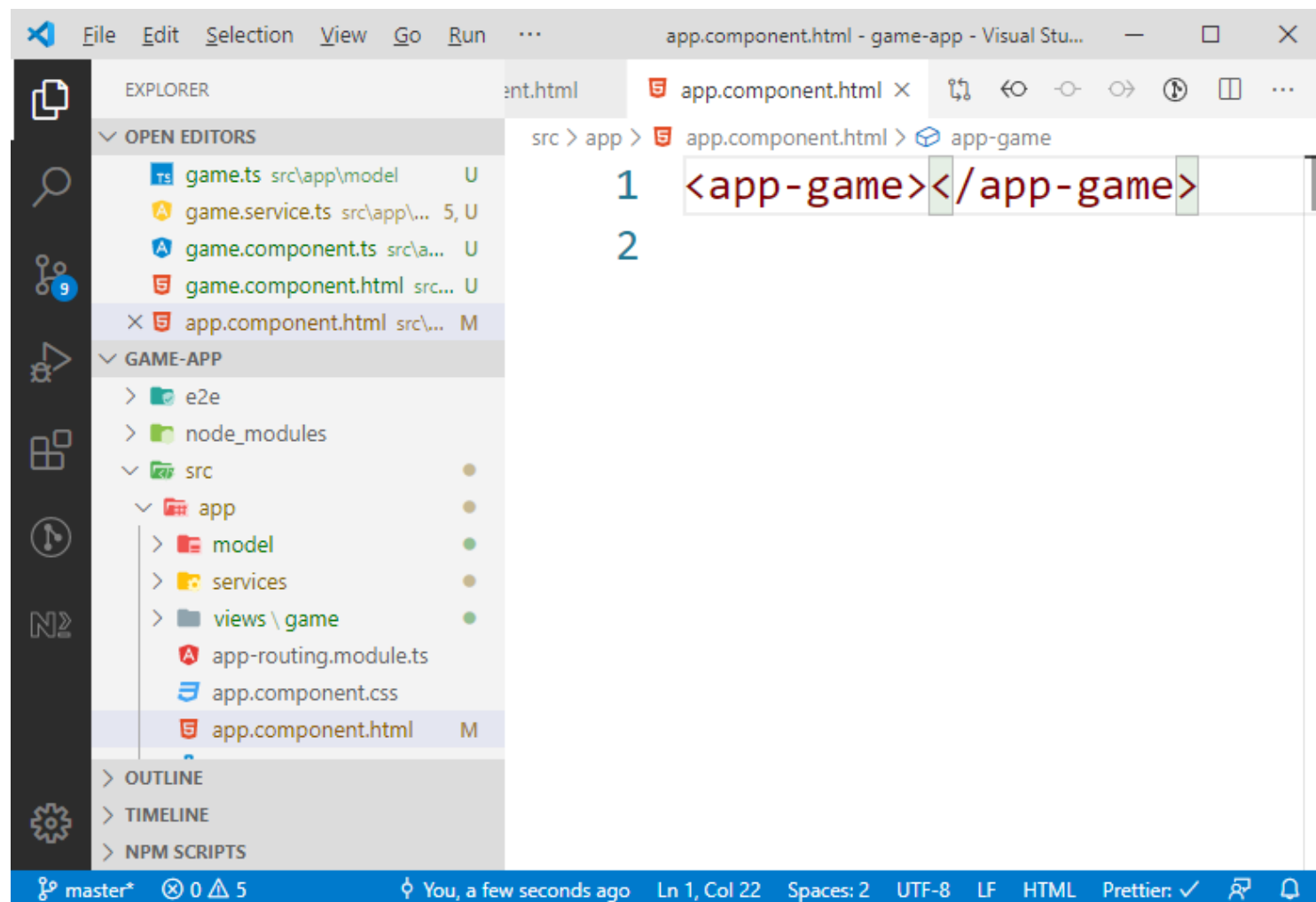
- Adicione uma chamada ao método criado no callback de exibição do componente (ngOnInit)

```
...  
  
ngOnInit(): void {  
    this.refreshGames();  
}  
  
...
```

Utilizando Componente

- Remova o conteúdo do componente principal (app-component) e adicione uma chamada ao componente de jogos:

Utilizando Componente de Jogos



Edição

- Crie uma nova propriedade no componente de jogos (game-component.ts)
- Ela representará o jogo selecionado para edição ou inserção

Edição

...

```
export class GameComponent implements OnInit {  
    games = new Array<Game>();  
  
    selGame?: Game = undefined;  
  
    constructor(private gameService: GameService) { }  
  
    ngOnInit(): void {  
    }  
}
```

...

Campos de Edição

- Utilize o elemento selGame em campos de edição na página (game-component.html) para criar uma área de edição, conforme a seguir:

Campos de Edição

```
<h2>Meus Jogos</h2>
<ul>
  <li *ngFor="let game of games">
    {{game.name}} - {{game.status}}
  </li>
</ul>

<div *ngIf="selGame">
  <input type="text" [(ngModel)]="selGame.name">
  <br>
  <select [(ngModel)]="selGame.status">
    <option value="na" label="Não Informado"></option>
    <option value="q" label="Quero"></option>
    <option value="t" label="Tenho"></option>
    <option value="j" label="Jogando"></option>
    <option value="z" label="Zerei"></option>
  </select>
  <br>
  <input type="text" [(ngModel)]="selGame.genre">
  <br>
  <input type="text" [(ngModel)]="selGame.platform">
</div>
```

FormsModule

- Não se esqueça de adicionar a importação ao `FormsModule` no `app-module.ts`, conforme a seguir:

FormsModule

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { GameComponent } from './views/game/game.component';
import { FormsModule } from '@angular/forms';

@NgModule({
  declarations: [
    AppComponent,
    GameComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Método de Seleção

- Faça um método de seleção de jogo (no game-component.ts) e associe-o ao evento de clique no HTML (game-component.html) dos itens de lista:

Método de Seleção

```
...  
    ngOnInit(): void {  
  
        }  
  
        selectGame(game: Game): void {  
            this.selGame = game;  
        }  
    ...
```

Método de Seleção

```
<h2>Meus Jogos</h2>
<ul>
  <li *ngFor="let game of games" (click)="selectGame(game);">
    {{game.name}} - {{game.status}}
  </li>
</ul>
```

Botões de Ação

- Crie botões de ação (no game-component.html), visando permitir a criação de um novo item, salvar um item e cancelar a ação atual
- Implemente também os respectivos métodos no ts (game-component.ts)

Botões de Ação

- O método de newGame deverá apenas criar um novo item
- O método cancel cancelar a criação ou seleção
- E o método de salvamento deverá chamar os respectivos métodos de alteração do serviço e atualizar a lista

Botões de Ação

- Também vamos precisar de um atributo de controle para saber se o usuário está editando ou inserindo um jogo novo (flag)
- Esse atributo será utilizado para decidir qual método chamar (insert ou update)

Botões de Ação

- Então crie também o atributo como um booleano e utilize-o no método selectGame para que ligue a flag como edição
- O método newGame deverá desligar a flag
- No final, todo o código ficará como a seguir

Botões de Ação

```
<br>  
  <input type="text" [(ngModel)]="selGame.genre">  
  <br>  
  <input type="text" [(ngModel)]="selGame.platform">  
  
  <button (click)="save();">Save</button>  
  <button (click)="cancel();">Cancel</button>  
  
</div>  
  
<button (click)="newGame();">New</button>
```

Botões de Ação

```
...
    editMode = false;
...
selectGame(game: Game): void {
    this.selGame = game;
    this.editMode = true;
}

newGame(): void {
    this.selGame = new Game('', '', '');
    this.editMode = false;
}

cancel(): void {
    this.selGame = undefined;
}

save(): void {
    if (!this.selGame) { return; }

    if (this.editMode) {
        this.gameService.update(this.selGame);
    } else {
        this.gameService.insert(this.selGame);
    }
    this.selGame = undefined;
    this.refreshGames();
}
...
```

Botão de Exclusão

- Crie um botão dentro do li para chamar um método de remoção do item com base no id
- No método de remoção, chame o serviço e solicite que remova o item pelo ID

Botão de Exclusão

```
...
<h2>Meus Jogos</h2>
<ul>
  <li *ngFor="let game of games" (click)="selectGame(game);">
    {{game.name}} - {{game.status}}
    <button (click)="remove(game.id);">x</button>
  </li>
</ul>
...
```

```
...
  remove(id: number): void {
    if (id === undefined) { return; }
    this.gameService.remove(id);
    this.refreshGames();
  }
...
```

INTEGRAÇÃO COM WEB STORAGE

Utilizando Web Storage

- Web Storage são dados armazenados localmente no navegador de um usuário
- Existem dois tipos de armazenamento:
 - Local Storage - dados sem data de validade que permanecem após o fechamento da janela do navegador
 - Session Storage - dados que são limpos após o fechamento da janela do navegador

Utilizando Local Storage

Útil para salvar dados como preferências do usuário (tema de cores claras ou escuras em um site), lembrar itens do carrinho de compras ou lembrar que um usuário está conectado a um site

Utilizando Local Storage

- Anteriormente, os cookies eram a única opção para armazenar dados temporários locais
- O armazenamento local tem um limite de armazenamento significativamente mais alto (5 MB vs 4KB) e não é enviado a cada solicitação HTTP, portanto pode ser uma opção melhor

Métodos do Local Storage

Método	Descrição
setItem()	Adicionar uma chave e um valor ao local storage
getItem()	Obtém de volta um valor através de uma chave
removeItem()	Remove um item pela chave
clear()	Limpa todos os dados

Utilizando Local Storage

- Para salvar dados, utilizamos o método `setItem`, especificando uma chave e um valor

```
localStorage.setItem('chave', 'valor');
```

Utilizando Local Storage

- Para recuperar dados salvos, utilizamos o método getItem, especificando a chave do item salvo

```
const item = localStorage.getItem('chave');
```

Salvando o Array no Local Storage

- Para nós, é necessário salvar vários dados no local storage (um array), não apenas um dado.

Como proceder?

- Podemos transformar o array em um JSON e salvá-lo como uma string

JSON

- Acrônimo para JavaScript Object Notation
- Formato compacto, de padrão aberto
independente, de troca de dados simples e
rápida (parsing) entre sistemas

JSON

- Utiliza texto legível a humanos, no formato chave-valor
- Modelo de transmissão de informações no formato texto, muito usado em web services (REST) e aplicações AJAX e substitui o uso do XML

JSON

- JSON vai criar uma representação de texto de nosso array, como a seguir:

JSON

```
[
  {
    "name":"Zelda",
    "status":"t",
    "genre":"Adventure",
    "platform":"Switch",
    "id":0
  },
  {
    "name":"Resident Evil 3",
    "status":"q",
    "genre":"Horror",
    "platform":"PC",
    "id":1
  }
]
```

JSON

- Para transformar um array em JSON, podemos utilizar o método stringify da classe JSON (do navegador)

```
JSON.stringify(array)
```

JSON

- Para transformar uma string JSON de volta em Array, podemos utilizar o método parse da classe JSON

```
JSON.parse(string)
```

Local Storage no Game App

- Para fazer com que o game app seja capaz de salvar e carregar o array do local storage, vamos adicionar dois métodos: o save e o load
- O save irá salvar o array no local storage, convertendo o em JSON antes
- E o load fará o carregamento de volta do LS

Local Storage no Game App

- O método de save será chamado sempre que uma alteração for feita, em todos os demais métodos de alteração
- O load será chamado na listagem, visando atualizar o array do serviço com os dados salvos

Local Storage no Game App

- Atualize o serviço com os métodos novos, como a seguir:

```
...
save(): void {
    localStorage.setItem('games', JSON.stringify(this.games));
    localStorage.setItem('gameAutoGeneratedId', this.autoGeneratedId.toString());
}

load(): void {
    const storageValues = localStorage.getItem('games');
    if (storageValues) {
        this.games = JSON.parse(storageValues)
    }
    else {
        this.games = new Array<Game>();
    }
    const autoGenId = localStorage.getItem('gameAutoGeneratedId');
    if (autoGenId) {
        this.autoGeneratedId = Number(autoGenId);
    }
}
...
```

Local Storage no Game App

- E atualize os métodos de salvamento/listagem, como a seguir:

Local Storage no Game App

...

```
insert(game: Game) {  
    game.id = this.autoGeneratedId;  
    this.games.push(game);  
    this.autoGeneratedId++;  
    this.save();  
}
```

```
list(): Array<Game> {  
    this.load();  
    return this.games;  
}
```

...

Local Storage no Game App

```
...
remove(id: number) {
  for (let i = 0; i < this.games.length; i++) {
    let m = this.games[i];

    if (m.id === id) {
      this.games.splice(i, 1);
      break;
    }
  }
  this.save();
}

update(game: Game) {
  for (let i = 0; i < this.games.length; i++) {
    let g = this.games[i];
    if (g.id === game.id) {
      this.games[i] = game;
      break;
    }
  }
  this.save();
}
...
```

ROTAS

Rotas do Angular

- O Angular trabalha com a composição de aplicações em página única
- Nesse cenário, os usuários permanecem em uma única página, mas a visualização dessa página muda dependendo do que o usuário faz

Rotas do Angular

- Para lidar com a navegação de uma view para outra, utilizamos o router (roteador) do Angular
- O router permite a navegação interpretando um URL do navegador como uma instrução para alterar a exibição de um componente

Rotas do Angular

- Quando um link for acionado e apontar para determinado endereço (por exemplo, opções de menu forem acionadas) determinado componente poderá ser carregado e exibido na página

Rotas do Angular

- O sistema de rotas precisa de alguns elementos para funcionar corretamente:
 - Um array contendo todas as definições de rotas (no `app-routing.module.ts`)
 - Um local onde os componentes acionados serão carregados (dentro de `app-component`), também chamado de `outlet`
 - Links que acionem as rotas (menu) no `app-component`

Array de rotas

- O array de rotas possui o seguinte formato:

```
const routes: Routes = [  
  { path: 'test1-component', component: Test1Component },  
  { path: 'test2-component', component: Test2Component },  
];
```


Array de rotas

- Onde **path** indica qual o caminho de URL deve ser digitado na barra de endereços (ou ser fornecido como href de um link) para carregar o componente e **component** corresponde ao componente que deve ser carregado

Array de rotas

- O array deve ser declarado dentro do elemento *app-routing.module.ts* e não precisa conter todos os componentes da aplicação, apenas os componentes que deseja-se permitir acesso por links ou pela URL

Array de rotas no app-routing.module

```
import { NgModule } from '@angular/core';  
import { Routes, RouterModule } from '@angular/router';  
import { Test1Component } from '../test1/test1.component';  
import { Test2Component } from '../test2/test2.component';
```

```
const routes: Routes = [  
  { path: 'test1-component', component: Test1Component },  
  { path: 'test2-component', component: Test2Component }  
];
```

```
@NgModule({  
  imports: [RouterModule.forRoot(routes)],  
  exports: [RouterModule]  
})  
export class AppRoutingModule { }
```

Router Outlet

- O router outlet corresponde ao local onde os componentes serão carregados quando a URL respectiva for acionada
- Utilizado através da tag **router-outlet**

Router Outlet

- Normalmente são definidos **dentro do app-component**, no local onde os elementos principais da aplicação deverão ser carregados (separado de itens de menu, layout e etc), como o *container* principal onde o conteúdo da aplicação ficará

Router Outlet

<router-outlet></router-outlet>

Links para componentes

- Os links vão disparar o acionamento do router para solicitar o carregamento de determinado componente
- Normalmente definidos no mesmo local onde encontra-se o router-outlet

Links para componentes

- Utilizamos links normais para links de rotas, com alguns atributos adicionais:

```
<a routerLink="/test1-component">Nome</a>
```


Links para componentes

- Onde routerLink corresponde a rota de componente configurada no app-routing.module

EXERCÍCIO

Exercício

- Implemente um CRUD de sua preferência (livros, cartas de magic, lista de compras, o que preferir)
- O item de modelo desse CRUD deve ter ao menos 5 atributos e não pode ser de um item que já fizemos até aqui (filmes, jogos)

Exercício

- O CRUD deve permitir gerenciar as informações e nome do item de negócio (inserir, excluir, alterar, listar)
- Melhore o CSS da aplicação e deixe-a mais apresentável

Exercício

- DESAFIO 1! Crie um segundo CRUD, que permita gerenciar um atributo do primeiro (por exemplo, tipo, categoria, fabricante, etc)

Exercício

- DESAFIO 2! Implemente também a funcionalidade de busca de itens por um dos atributos do modelo

"That's all Folks!"