Universidade Federal de Pernambuco
Centro de Informática

Graduação em Engenharia da Computação

Estudo comparativo entre ferramentas de teste unitário para Angular ao programar com o paradigma reativo utilizando RxJS

Aluno: André Luiz Figueirôa de Barros (alfb@cin.ufpe.br)
Orientador: Leopoldo Motta Teixeira (lmt@cin.ufpe.br)
Área: Engenharia de Software

Recife, 2023

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Barros, André Luiz Figueirôa de.
Estudo comparativo entre ferramentas de teste unitário para Angular ao programar com o paradigma reativo utilizando RxJS / André Luiz Figueirôa de Barros. - Recife, 2023.
46 : il., tab.

Orientador(a): Leopoldo Motta Teixeira
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Engenharia da Computação - Bacharelado, 2023.
Inclui referências, apêndices.

1. Engenharia de Software. 2. Frontend. 3. Testes Unitários. I. Teixeira, Leopoldo Motta. (Orientação). II. Título.

000 CDD (22.ed.)

RESUMO

Angular é um *framework frontend*, construído em *TypeScript*, baseado em componentes para construir aplicações *web*.

Parte da comunidade do *Angular* defende o uso do paradigma de programação reativa para ter ganhos em manutenibilidade. No *Angular* isso pode ser alcançado com o uso da biblioteca *RxJS*, porém o código da aplicação pode tornar-se mais complexo.

Independentemente do paradigma de programação escolhido, os testes são parte fundamental no ciclo de desenvolvimento de *software* quando se busca atingir níveis altos de qualidade. O intuito desse trabalho é comparar três estratégias que podem ser utilizadas para fazer testes unitários em uma aplicação *Angular* que utiliza programação reativa com *RxJS*, sendo elas: O uso do método *subscribe* dos *Observables* do *RxJS*, o uso da *API* de *Marble Diagrams* que também é fornecida pelo *RxJS* e o uso da biblioteca *Observer-spy* que foi criada com o intuito de facilitar os testes de *Observables* do *RxJS*.

As estratégias utilizadas foram comparadas com as seguintes métricas: quantidade de linhas, legibilidade do teste e finalidade das asserções. O objetivo da comparação é identificar quais das estratégias são melhores e piores em diversos cenários comumente encontrados no desenvolvimento de *software frontend*.

Palavras-chaves: Frontend, Testes, Angular, RxJS, Programação reativa.

ABSTRACT

Angular is a frontend framework, built in TypeScript, based on components for building web applications.

Part of the Angular community advocates for the use of reactive programming paradigm to achieve gains in maintainability. In Angular, this can be achieved using the RxJS library, but the source code becomes more complex.

Regardless of the chosen programming paradigm, testing is a fundamental part of the software development cycle that aims to achieve high levels of quality. The purpose of this work is to compare three strategies that can be used for unit testing in an Angular application that uses reactive programming with RxJS, namely: using the subscribe method of RxJS Observables, using the Marble Diagrams API also provided by RxJS, and using the Observer-spy library that was created to facilitate testing of RxJS Observables.

The strategies used were compared with the following metrics: amount of lines, test readability, and purpose of assertions. The objective of the comparison is to identify which strategies are better and worse in various scenarios commonly encountered in frontend software development.

Keywords: Frontend, Tests, Angular, RxJS, Reactive programming.

LISTA DE TABELAS

LISTA DE FIGURAS

LISTA DE CÓDIGOS

SUMÁRIO

1. INTRODUÇÃO

1.1 Contexto e motivação

Uma aplicação *web* consiste em um *software* que executa em um navegador *web*, que se comunica com um ou mais servidores para troca e manipulação de dados. O *software* que executa no navegador da máquina do usuário é chamado de *frontend* e é responsável pela interface visual do sistema e pela interface com o usuário da aplicação. Já o código que executa nos servidores e que são responsáveis por gerenciar os dados da aplicação é chamado de *backend* [1].

Testes de *software* são um processo para avaliar se um *software* funciona da forma desejada. Os testes são feitos para encontrar falhas no sistema, de forma que elas sejam corrigidas antes que os usuários do sistema usem a aplicação [2]. Existem diversos tipos de teste de *software*, sendo o mais básico deles os testes unitários que são responsáveis por testar métodos, classes, componentes ou módulos de forma independente do resto do sistema [3].

O paradigma de programação reativa é baseado na criação de fluxo de dados e na observação desses fluxos para reagir a mudanças no mesmo [4, 5].

O *Angular* é um *framework* para desenvolvimento *frontend* no contexto de aplicações *web* [6]. Ele é um dos *frameworks* mais utilizados no universo de desenvolvimento *frontend*, ocupando a 2ª posição como *framework frontend* mais utilizado no levantamento feito pelo *State of JS* de 2022 [7].

O *Angular* utiliza *RxJS* [8] internamente, uma biblioteca que é utilizada para programar de forma reativa. Por estar integrada com o *framework,* o *RxJS* se tornou bastante popular dentro da comunidade *Angular*, a ponto de gerar discussões sobre basear toda uma aplicação sobre o uso de programação reativa com *RxJS* [9, 10].

Este trabalho tem como objetivo fazer um estudo comparativo entre 3 ferramentas que podem ser utilizadas para realizar testes unitários em uma aplicação *frontend* desenvolvida em *Angular* que utiliza o paradigma de programação reativa baseada em *RxJS* [8]. A motivação para realizar esse estudo foi o trabalho “Estudo comparativo entre ferramentas de teste para React” [11].

O entendimento do que é e como funciona o RxJS é fundamental para a compreensão desse trabalho, segundo a própria documentação a biblioteca se define como: “*ReactiveX combines the Observer pattern with the Iterator pattern and functional programming with collections to fill the need for an ideal way of managing sequences of events*” [8]. Na prática a biblioteca define classes que são utilizadas na

criação e gerenciamento de fluxos de dados, sendo a mais importante delas a classe *Observable*.

A primeira ferramenta utilizada para realizar os testes unitários é a própria biblioteca do *RxJS*. Nesses testes, o método **`subscribe`** é chamado nos objetos da classe **`Observable`**, que representam os fluxos de dados, para ler os dados que serão utilizados para fazer as asserções dos testes.

A segunda ferramenta utiliza *Marble Diagrams* para realizar os testes. Estes diagramas representam uma forma de visualizar os fluxos de dados e são utilizados com uma *API* específica para testes disponibilizada pela própria biblioteca do *RxJS* [12]. Com essa solução é possível comparar os fluxos de dados inteiros entre si.

A terceira estratégia recorre ao uso de outra biblioteca, a **`observer-spy`**, que foi criada com o propósito de facilitar os testes de código feito com o *RxJS* [13].

1.2 Objetivos gerais

O objetivo do trabalho é comparar 3 ferramentas diferentes que podem ser utilizadas para realizar testes unitários em uma aplicação feita em *Angular* com programação reativa utilizando *RxJS*.

A primeira ferramenta é o próprio *RxJS* e utiliza o método **`subscribe`** da classe **`Observable`** para leitura de dados. A segunda ferrramenta é a *API* de testes baseada em *marble diagrams* também fornecida pela biblioteca do *RxJS*. Já a terceira ferramenta é a biblioteca **`observer-spy`** criada para facilitar os testes de código *RxJS*.

O resultado dessa comparação é de interesse não apenas para profissionais que usam *Angular* como *framework* como também pode interessar aos profissionais que trabalham com *frontend* de maneira geral o *Angular* é uma das tecnologias mais utilizadas do mercado. Além disso, o debate sobre reatividade não é exclusivo do *Angular*.

1.3 Objetivos específicos

- Implementação de uma aplicação *web* com *frontend* desenvolvido em *Angular* para realização dos testes.
- Escrever testes análogos utilizando as 3 estratégias definidas citadas na Seção 3.1.

- Utilizar a cobertura de testes para certificar que os testes feitos com cada uma das ferramentas são análogos
- Comparar os testes feitos utilizando as diferentes estratégias utilizando as métricas definidas a seguir com o fim de apontar quais dos métodos é mais apropriado para cada situação.
    - Quantidade de linhas
    - Legibilidade do código
    - Finalidade das asserções

Essas métricas foram escolhidas com base no trabalho utilizado como referência [11]. Outras métricas que também poderiam ser utilizadas mas foram descartadas são tempo de execução e uso de memória, pois seria necessário uma aplicação de grande para ter uma real diferença nesses valores, o que não foi viável devido ao tempo e escopo do projeto.

1.4 Seções do documento

A Seção 2 apresenta os conceitos fundamentais para o entendimento do projeto.

A Seção 3 apresenta as 3 abordagens avaliadas neste trabalho e o código do projeto desenvolvido.

A Seção 4 apresenta os códigos desenvolvidos para testar a aplicação e compara as 3 abordagens.

A Seção 5 encerra o trabalho mostrando as conclusões obtidas com o final do estudo e aponta uma sugestão para trabalho futuro.

- **2. REVISÃO DA LITERATURA**

Os conceitos fundamentais para o entendimento do projeto são apresentados nesta seção.

2.1 Conceitos

2.1.1 *Angular*

Angular é um *framework* construído em *TypeScript*, baseado em componentes para construir aplicações *web*. O *framework* é formado por várias bibliotecas que fornecem diversas funcionalidades, como gerenciamento de formulário, roteamento, comunicação cliente-servidor, e muitas outras [6].

2.1.1.1 Componentes

As principais peças de uma aplicação *Angular* são os componentes [14]. Eles representam fragmentos da UI de uma aplicação e são compostos por:

- Um arquivo *HTML* que representa a visão daquele componente. É nesse arquivo que são definidos os elementos visíveis [14].
- Um arquivo *TypeScript* com uma declaração de classe que representa o modelo daquele componente. É nessa classe que é definido o comportamento do componente. É também nesse arquivo que é definido o seletor *CSS* do componente para que ele possa ser utilizado por outros componentes [14].
- Um arquivo de estilização opcional, utilizado para aplicar estilos ao arquivo *HTML* do componente [14].

2.1.1.2 Serviços

Serviços são classes com objetivos bem definidos que podem ser acessados por outros elementos do sistema através do sistema de injeção de dependência nativo do *Angular*. Os serviços normalmente são utilizados para reduzir as responsabilidades dos componentes, tornando o código mais modularizado e reutilizável [15].

2.1.1.3 *Angular CLI*

Como o próprio nome já diz, *Angular CLI* trata-se de uma interface de linha de comando para o *Angular*. Com essa ferramenta é possível criar um projeto *Angular*, criar componentes, criar serviços, criar módulos, executar testes, e muitas outras funcionalidades através de comandos em um terminal [16].

2.1.2 Testes unitários

Teste unitário é o tipo de teste mais baixo nível de um *software* e está bastante próximo do código fonte. Esse tipo de teste consiste em testar as classes, componentes e módulos da aplicação através de suas funções e métodos. Os testes unitários são os mais baratos de se automatizar e normalmente executam rapidamente [3].

2.1.2.1 Testando Componentes

Como explicado na Seção 2.1.1.1 o componente é uma combinação de um *template* HTML com uma classe em *TypeScript*. Apesar de ser possível testar a classe isoladamente, o que torna os testes mais simples, o teste se torna mais completo quando ele verifica o funcionamento da classe do componente trabalhando em conjunto com o seu template [17].

2.1.2.2 Testando Serviços

Como explicado na Seção 2.1.1.2 os serviços são compostos por apenas uma classe. Os testes unitários do mesmo se resumem a testar seus métodos e funções [18].

2.1.3 *Test Runner*

Test runner é um *software* que se responsabiliza pela execução dos testes e exportação dos resultados.[19].

2.1.3 *Jasmine* e *Karma*

O *Karma* é um *test runner*, ele é instalado por padrão quando se cria um novo projeto *Angular* através da *CLI* [20].

O *Jasmine* é um *framework* de teste, e assim como o *Karma* ele também é instalado por padrão quando se cria um projeto através do *Angular CLI* [20].

2.1.4 Programação Imperativa

No paradigma de programação imperativa o programador passa uma sequência de comandos para a máquina para que ela possa desempenhar sua função. Nesse paradigma as instruções são passadas para o computador como um passo a passo [21].

2.1.5 Programação Declarativa

Se no paradigma de programação imperativa o desenvolvedor deve descrever *como* o computador deve fazer determinada tarefa, no paradigma de programação declarativa o desenvolvedor deve descrever *o quê* deve ser feito. Ou seja, o foco está no resultado da tarefa e não em como ela é executada [22].

2.1.6 Programação Reativa

A programação reativa é um tipo de programação declarativa que foca no uso de fluxo de dados e em como novos valores em cada um dos fluxos impactam o resto do programa [4, 5].

O código 2.1 foi escrito em *JavaScript*, o resultado desse código imprime na tela a mensagem “Olá Pessoa”. Mesmo ao alterar o valor da variável **`b`**, nenhum efeito vai se refletir no valor da variável **`c`**. Esse é um exemplo de programação imperativa. Se pensarmos em **`a`**, **`b`** e **`c`** como fluxos de dados ao invés de variáveis tradicionais do *JavaScript*, uma mudança no valor de **`b`** seria automaticamente refletida no valor de **`c`**, resultando a mensagem “Olá André”.

Código 2.1 - Exemplo código imperativo

```
let a = 'Olá';
let b = ' Pessoa';
let c = a + b;
b = ' André';

console.log(c)
```

Fonte: Elaborado pelo autor, 2023.

2.1.7 *RxJS*

RxJS é uma biblioteca que fornece as ferramentas necessárias para a criação e manuseio de eventos. Seus conceitos assim como um guia inicial de como usar essa ferramenta encontram-se na documentação oficial em [8].

2.1.8 *NgRx*

NgRx é um conjunto de bibliotecas que auxiliam no desenvolvimento de aplicações reativas em *Angular* [23]. Esse projeto utilizou as seguintes bibliotecas:

- **`@ngrx/store`** - Utilizada para gerenciamento de estado global da aplicação
- **`@ngrx/effects`** - Utilizada para registrar efeitos colaterais ao interagir com o estado global da aplicação
- **`@ngrx/component-store`** - Utilizada para gerenciamento de estado local da aplicação

As 3 bibliotecas acima são baseadas no *RxJS*. Os conceitos de cada uma delas assim como um guia contendo exemplo de uso encontram-se na documentação oficial [23].

2.2 Trabalhos relacionados

Em 2021, o trabalho de graduação “Estudo comparativo entre ferramentas de teste para React” [11] por Ferreira, Pamella compara *Enzyme* e *React Testing Library* (RTL), duas ferramentas utilitárias para desenvolver testes em aplicações *web* feitas em *React*. Neste trabalho a autora conclui que apesar de ambas as ferramentas empatarem na cobertura de teste, e os testes possuírem em média praticamente a mesma quantidade de linhas, os testes do RTL eram mais legíveis e simulavam a interação do usuário com a aplicação, enquanto que o *Enzyme* era mais voltado para testar o comportamento interno dos componentes [11].

Em 2016, o artigo “Comparison of Angular JS framework testing tools” [24] publicado por Nina Fat, Marijana Vujovic, Istvan Papp e Sebastian Novak compara o *Karma* e o *Protractor* no *AngularJS* para avaliar qual o mais apropriado em achar problemas de otimização. O trabalho conclui que o *Karma* se saiu melhor nesse contexto [24].

Assim como os trabalhos citados acima, esse trabalho faz um estudo comparativo entre ferramentas que podem ser utilizadas em testes de aplicações *web*. A diferença é que o estudo é feito em um contexto diferente: aplicações *web* feitas em *Angular* utilizando de programação reativa com *RxJS*.

3 DESENVOLVIMENTO DO PROJETO

3.1 Ferramentas de teste

3.1.1 *RxJS*

A primeira ferramenta utilizada para testar a aplicação é o próprio *RxJS*. Nessa abordagem é utilizado o método **`subscribe`** da classe **`Observable`** para ler o valores existentes nos fluxos de dados da aplicação. Após a leitura dos valores é possível fazer as devidas asserções. No escopo desse trabalho essa abordagem será chamada de *subscribing strategy*.

3.1.2 *Marble Diagrams*

Além das ferramentas para criação e manipulação de *Observables,* a biblioteca *RxJS* também fornece ferramentas para testá-los focada em diagramas de *Marble* [12]. Essa sub-biblioteca de testes fornecida pelo *RxJS* permite a criação de *Observables* a partir de um diagrama de *Marble*, manipulação de um relógio virtual para simular a passagem do tempo nos testes e comparação entre *Observables*. Nessa abordagem é possível comparar fluxos de dados como um todo. No escopo desse trabalho essa abordagem será chamada de *marble strategy*.

3.1.3 *observer-spy*

A biblioteca **`observer-spy`** foi criada especificamente para testar *Observables* do *RxJS* [13]. Essa abordagem resume-se a utilizar as funções disponíveis nessa biblioteca para realizar os testes.

3.2 Aplicação desenvolvida

Devido ao escopo do trabalho, não foi viável o desenvolvimento de um projeto complexo como uma aplicação empresarial, *e-commerce* ou rede social. Contudo, o projeto desenvolvido, apesar de simples, implementa funcionalidades comuns à esses tipos de aplicações como:

- Listagem e filtragem de dados vindo de uma *API REST*
- Paginação
- Gerenciamento de estado global
- Gerenciamento de estado local

No escopo desse trabalho essa abordagem será chamada de *observer-spy strategy*.

3.2.1 *Backend*

O *backend* desenvolvido utiliza dados que foram obitidos da *PokeAPI* [25], uma *RestAPI* pública feita para fins educacionais que disponibiliza dados referentes aos jogos da franquia *Pokémon*.

De acordo com a documentação da *PokeAPI* [25], não existe um *endpoint* que retorne a lista com as informações completas de cada *Pokémon*. O que existe é um *endpoint* que retorna uma lista de recursos, ou seja, ao invés de cada item da lista conter as informações de um *Pokémon*, cada item na lista contem apenas duas informações:

- **`name`**: O nome do Pokémon
- **`url`***:* A *URL* para o *endpoint* que contém as informações daquele *Pokémon*.

Devido a essa característica, para adquirir informações de *N Pokémon* seria necessário fazer *N+1* requisições:

- 1 requisição para obter a lista de recursos
- *N* requisições, sendo 1 para receber os dados de cada *Pokémon* da lista

Foi desenvolvido um servidor que fez estas requisições uma única vez, salvando assim localmente as informações de todos os *Pokémon* disponíveis na *PokeAPI*. Isso foi feito para não quebrar as políticas de uso justo da *PokeAPI* [25], que solicita que os usuários salvem localmente os dados para evitar um alto volume de requisições.

Além disso, essa solução foi conveniente pois remove do *frontend* a necessidade de gerenciar essas requisições.

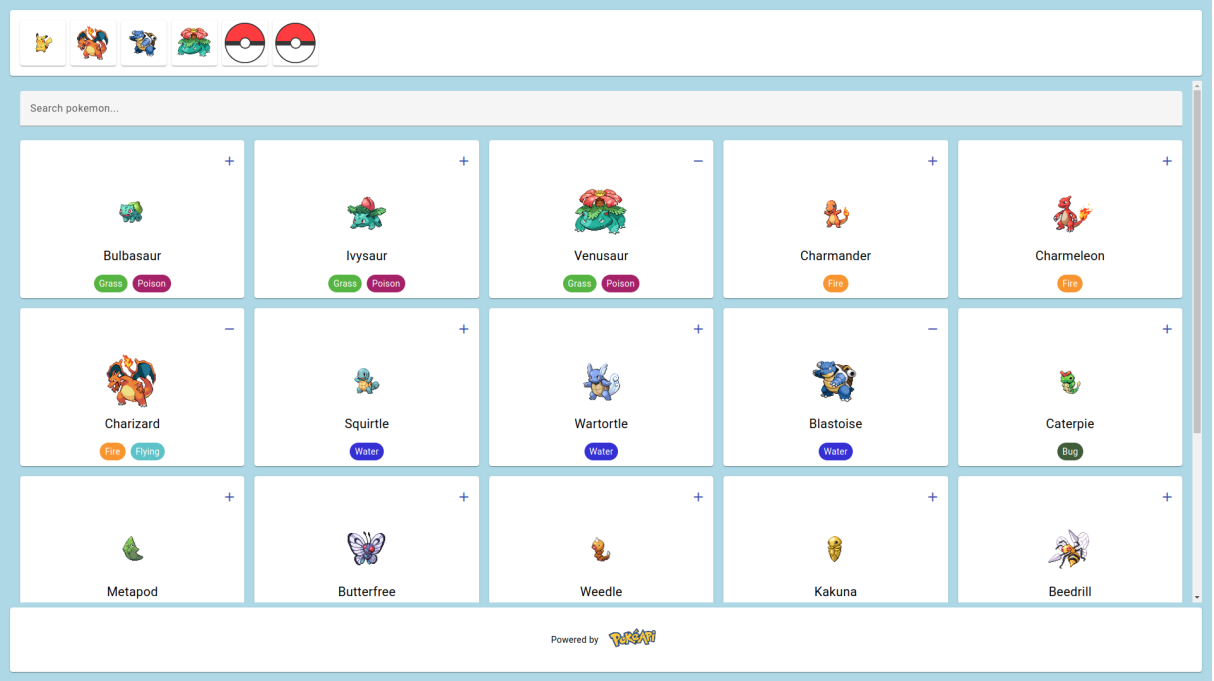
O código do *backend* foi desenvolvido pelo autor deste trabalho e encontra-se disponível em [26], assim como suas instruções de uso.

3.2.2 *Frontend*

A aplicação *frontend* consiste em uma página capaz de listar *Pokémon*, os dados de *Pokémon* vem da aplicação *backend* detalhada na seção anterior. A apliação também é capaz de buscar por *Pokémon* através de seus nomes. Além disso, o usuário também é capaz de formar um time de até 6 *Pokémon*.

O código *frontend* também foi desenvolvido pelo autor deste trabalho e encontra-se disponível em [27], assim como suas instruções de uso.

Figura 3.1 - Tela da aplicação



Fonte: Elaborado pelo autor, 2023.

3.2.2.1 Dependências

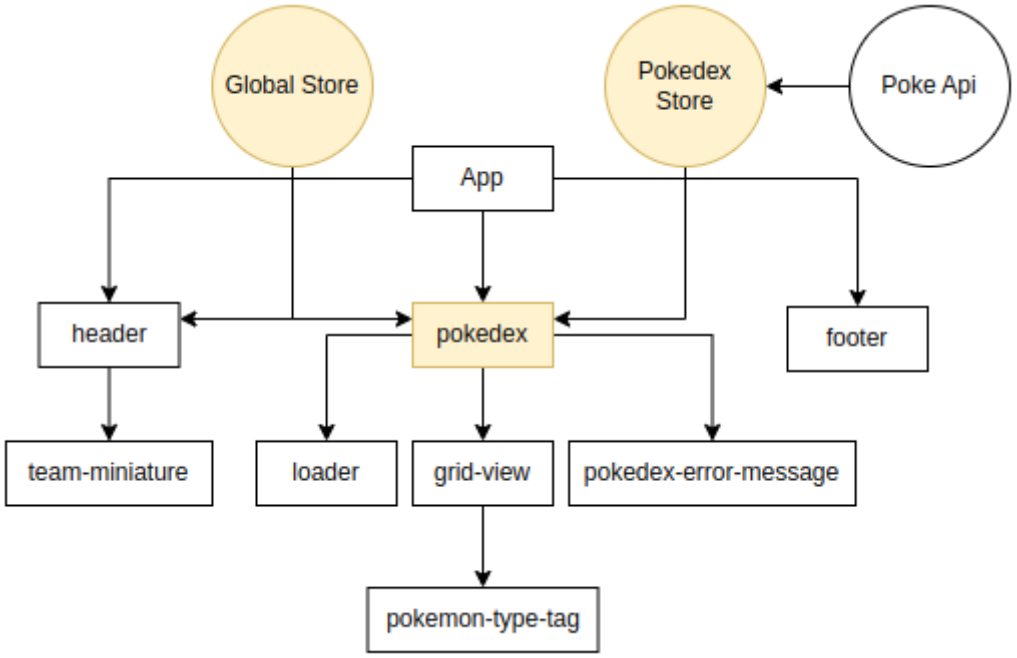
Algumas dependências bastante utilizadas no ecossistema *Angular* foram adicionadas ao projeto:

- *Angular Material*: Uma biblioteca de UI que possui diversos componentes disponíveis para usar em aplicações *Angular* [28]
- *Ngrx*: Bibliotecas para gerenciamento de estado em aplicações reativas feitas com *Angular*. Mais detalhes na Seção 2.1.8.

3.2.2.2 Estrutura

A aplicação possui a estrutura representada pela Figura 3.2. Os retângulos representam os componentes e os círculos representam serviços. Em amarelo encontram-se os componentes e serviços que foram testados usando as abordagens escolhidas por esse trabalho.

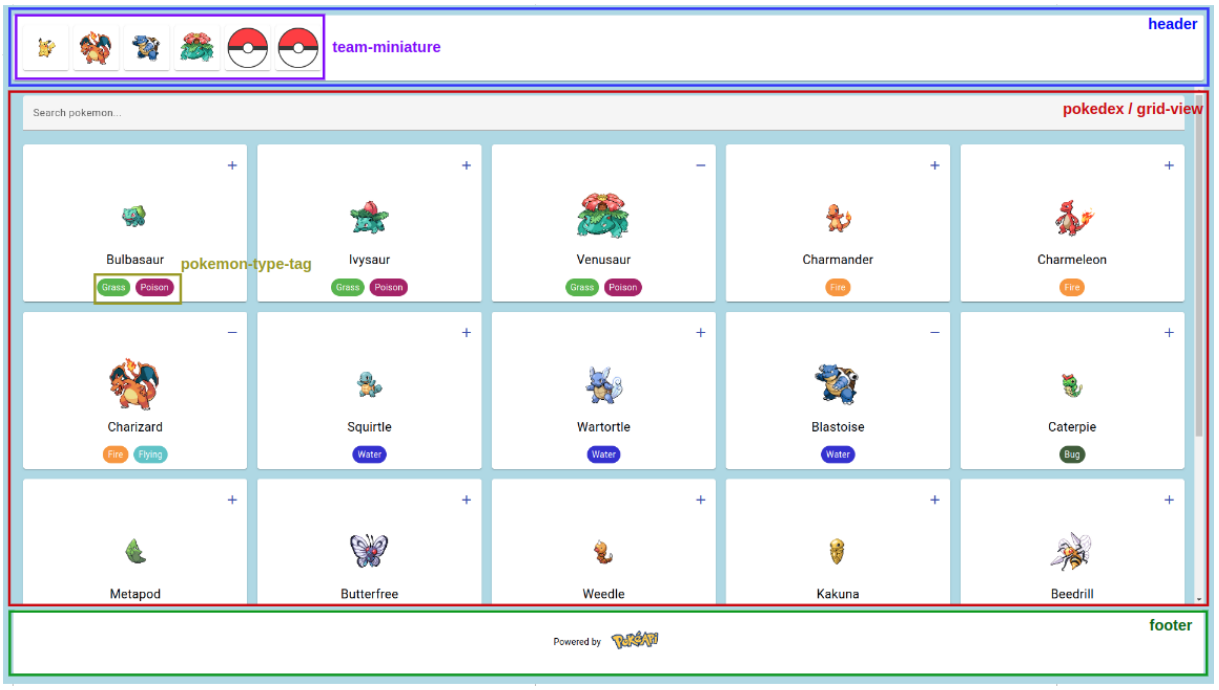
Figura 3.2 - Estrutura do projeto



Fonte: Elaborado pelo autor, 2023.

A Figura 3.3 mapeia os componentes mostrados na Figura 3.2 com a tela da aplicação representada pela Figura 3.1, com exceção dos componentes **`loader`** e **`pokedex-error-message`** que não estão renderizados na figura pois representam estados de carregamento e de erro respectivamente.

Figura 3.3 - Estrutura do projeto vs Tela da aplicação



Fonte: Elaborado pelo autor, 2023.

4. TESTES E COMPARAÇÕES

4.1 Alvo dos testes

A Figura 3.2 mostra a estrutura da aplicação baseada em seus componentes e serviços, apenas os elementos com o fundo da cor amarela foram utilizados para comparação dos testes.

Os componentes que não foram testados tem apenas a responsabilidade de renderizar, logo não possuem lógica que faça sentido ser testada por alguma das abordagens escolhidas.

O serviço *Poke Api* é responsável por configurar requisições *HTTP*, utilizando o *HttpClient* que faz parte do *framework Angular* (Código 4.1). Para esse cenário específico a documentação do *Angular* é bem direta quanto a forma na qual esse código deve ser testado [29].

Código 4.1 - poke-api.service.ts

```
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';
import { ListResult } from '../models/list-result.model';
import { Pokemon } from '../models/pokemon.model';

export const LIST_POKEMON_URL = 'http://localhost:3000/pokemon';

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

 constructor(private http: HttpClient) {}

 getPokemonList(page: number = 0, searchText: string = ''):
Observable<ListResult<Pokemon>> {
   return this.http.get<ListResult<Pokemon>>(LIST_POKEMON_URL, {
     params: {
       page,
       searchText
     }
   });
 }

}
```

Fonte: Elaborado pelo autor, 2023.

4.2 *Pokedex Store*

Esse serviço é utilizado para o gerenciamento de estado local do componente *pokedex* utilizando as ferramentas disponíveis pela biblioteca **`@ngrx/component-store`**, citada na Seção 3.1.8. No Código 4.2 a classe **`PokedexStore`** estende a classe genérica **`ComponentStore`** que por sua vez é importada de **`@ngrx/component-store`**.

As responsabilidades desse serviço são as seguintes:

- Armazenar o estado do componente **`Pokedex`**
- Disponibilizar o estado do componente **`Pokedex`** de forma reativa
- Disponibilizar métodos para que seja possível atualizar o estado atual
- Reagir a mudança de estados e utilizar de forma reativa o serviço **`PokeApi`** para fazer requisições de mais *Pokémon*

O estado inicial é configurado na chamada no método **`super`** do **`constructor`** e possui o tipo da interface **`PokedexState`**, que é o tipo especificado ao estender a classe **`ComponentStore`**.

A interface **`PokedexState`**, possui 3 propriedades:

- **`pokemonList`**: responsável por armazenar a lista de Pokémon que deve ser renderizada pelo componente **`PokedexComponent`** e seus filhos
- **`requestStatus`**: pode assumir os valores **`pending`**, **`processing`**, **`success`** ou **`error`** e é utilizada para sinalizar o estado atual da requisição feita ao *backend*. A depender do estado, a aplicação pode mostrar os dados atuais, mostrar uma animação de carregamento ou exibir uma mensagem de erro.
- **`apiTrigger`**: essa propriedade é utilizada para disparar requisições feitas ao *servidor* quando modificada. Ela possui o tipo **`ApiEffectTrigger`** que também está definido no Código 4.2 e possui as seguintes propriedades:
  - **`currentPage`**: página atual da lista de *Pokémon*
  - **`lastPage`**: booleano que indica se está ou não na última página da lista de *Pokémon*
  - **`searchText`**: utilizado como parâmetro para fazer a filtragem de *Pokémon* por seus nomes
  - **`requestRetryCount`**: utilizado para contar quantas tentativas foram feitas caso a requisição para o *backend* falhe

O serviço disponibiliza para o componente apenas os estados **`pokemonList`**, **`requestStatus`** e **`lastPage`** através dos *Observables* **`pokemonList$`**, **`requestStatus$`** e **`lastPage$`** respectivamente (acima do **`constructor`***)*. Sendo as demais propriedades do estado utilizadas apenas internamente pelo serviço.

O método **`fetchPokemons`** configura um efeito colateral que, quando executado faz uma requisição ao servidor caso a aplicação não esteja na última página da lista. Em caso de sucesso ele atualiza a lista de *Pokémon*, em caso de falha uma mensagem é exibida ao usuário. No **`constructor`** esse método é chamado e configurado para reagir às mudanças que ocorrem no estado **`apiTrigger`**, sendo assim, executado de forma reativa toda vez que uma mudança for detectada.

O método **`resetRequestStatus`** é utilizado para resetar o estado da requisição para **`pending`** 1 segundo após a sua chamada. Ele é chamado pelo método **`fetchPokemons`** após a requisição resultar em falha ou sucesso.

Os demais métodos são utilizados apenas para modificar o estado atual do serviço. Dependendo de como o estado for alterado, o componente **`Pokedex`** e o efeito colateral **`fetchPokemons`** podem reagir a essa mudança.

Código 4.2 - pokedex.store.ts

```
import { Injectable } from "@angular/core";
import { ComponentStore } from "@ngrx/component-store";
import { catchError, concatMap, delay, distinctUntilChanged, EMPTY,
filter, map, Observable, tap } from "rxjs";
import { isEqual } from 'lodash-es';
import { PokeApiService } from
"../shared/data-access/poke-api.service";
import { ListResult } from "../shared/models/list-result.model";
import { Pokemon } from "../shared/models/pokemon.model";
import { RequestStatus } from "../shared/models/request-status.model";
import { MatSnackBar } from "@angular/material/snack-bar";

export interface PokedexState {
 pokemonList: Pokemon[];
 requestStatus: RequestStatus;
 apiTrigger: ApiEffectTrigger;
}
```

```
interface ApiEffectTrigger {
 currentPage: number;
 lastPage: boolean;
 searchText: string;
 requestRetryCount: number;
};

@Injectable()
export class PokedexStore extends ComponentStore<PokedexState> {
 readonly pokemonList$ = this.select(state => state.pokemonList);
 readonly requestStatus$ = this.select(state => state.requestStatus);
 private readonly fetchData$ = this.select(state => state.apiTrigger);
 readonly lastPage$ = this.fetchData$.pipe(map(data => data.lastPage));

 constructor(
   private readonly pokeApi: PokeApiService,
   private readonly snackBar: MatSnackBar
 ) {
   super({
     pokemonList: [],
     requestStatus: 'pending',
     apiTrigger: {
       requestRetryCount: 0,
       currentPage: 0,
       lastPage: false,
       searchText: ''
     }
   });
   this.fetchPokemons(this.fetchData$.pipe(distinctUntilChanged((a,b)
=> isEqual(a,b))));
 }

 readonly loadNextPage = this.updater(state => ({
   ...state,
   apiTrigger: {
     ...state.apiTrigger,
     requestRetryCount: 0,
     currentPage: state.apiTrigger.currentPage + 1
   }
 }));

 readonly searchPokemon = this.updater((state, searchText: string) =>
```

```
({
   ...state,
   pokemonList: [],
   apiTrigger: {
     requestRetryCount: 0,
     currentPage: 0,
     lastPage: false,
     searchText: searchText
   }
 }));

 readonly retryLastRequest = this.updater(state => ({
   ...state,
   apiTrigger: {
     ...state.apiTrigger,
     requestRetryCount: state.apiTrigger.requestRetryCount + 1
   }
 }));

 private readonly fetchPokemons = this.effect((data$:
Observable<ApiEffectTrigger>) => data$.pipe(
   filter(fetchData => !fetchData.lastPage),
   tap(() => this.updateRequestStatus('processing')),
   concatMap(fetchData =>
this.pokeApi.getPokemonList(fetchData.currentPage,
fetchData.searchText)
     .pipe(
       tap({
         next: result => {
           if (fetchData.currentPage === 0) {
             this.setPokemonList(result);
           } else {
             this.addPokemonsToList(result);
           }
         },
         error: err => {
           this.snackBar.open(err.message, 'Dismiss', {
horizontalPosition: 'end', verticalPosition: 'top'});
           this.updateRequestStatus('error');
         },
         finalize: () => this.resetRequestStatus()
       }),
       catchError(err => EMPTY)
```

```
      )),
  ));

  private resetRequestStatus = this.effect($ => $.pipe(
    delay(1000),
    tap(() => this.updateRequestStatus('pending'))
  ));

  private updateRequestStatus = this.updater((state, requestStatus:
RequestStatus) => ({
    ...state,
    requestStatus
  }));

  private addPokemonsToList = this.updater((state, pokemonListResult:
ListResult<Pokemon>) => ({
    ...state,
    pokemonList: [...state.pokemonList, ...pokemonListResult.content],
    requestStatus: 'success',
    apiTrigger: {
      ...state.apiTrigger,
      lastPage: pokemonListResult.last
    }
  }));

  private setPokemonList = this.updater((state, pokemonListResult:
ListResult<Pokemon>) => ({
    ...state,
    pokemonList: pokemonListResult.content,
    requestStatus: 'success',
    apiTrigger: {
      ...state.apiTrigger,
      lastPage: pokemonListResult.last
    }
  }));
  }
```

Fonte: Elaborado pelo autor, 2023.

4.2.1 Testes

Os seguintes testes foram desenvolvidos utilizando as 3 abordagens:

- *it should be able to fetch first page of pokemons on initialization*
- *it should be able to fetch more pages of pokemons when calling loadNextPage when not in last page*

- *it should not be able to fetch more pages of pokemons when calling loadNextPage when in last page*
- *it should be able to fetch data by search text*
- *it should be able to retry last request using same parameters, but incrementing the retry count by 1*
- *it should be able to reset requestStatus to pending after 1 second*

Comparações a respeito dos testes, mostrando trechos de código, são feitas na Seção 4.5.

4.3 *Global State*

É utilizado para o gerenciamento de estado global da aplicação, utilizando a biblioteca **`@ngrx/store`**, citada na Seção 3.1.8. O estado global é configurado no módulo raiz da aplicação, como pode ser visto no Código 4.3.

Código 4.3 - app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from
'@angular/platform-browser/animations';
import { HttpClientModule } from '@angular/common/http';
import { HeaderComponent } from './header/header.component';
import { FooterComponent } from './footer/footer.component';
import { StoreModule } from '@ngrx/store';
import { MatSnackBarModule } from '@angular/material/snack-bar';
import { teamReducer } from './shared/state/team/team.reducer';
import { EffectsModule } from '@ngrx/effects';
import { TeamEffects } from './shared/state/team/team.effects';

@NgModule({
 declarations: [
   AppComponent
 ],
 imports: [
   BrowserModule,
   AppRoutingModule,
   BrowserAnimationsModule,
   HttpClientModule,
   HeaderComponent,
   FooterComponent,
   StoreModule.forRoot({
```

```
      team: teamReducer
    }),
    EffectsModule.forRoot([TeamEffects]),
    MatSnackBarModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Fonte: Elaborado pelo autor, 2023.

O estado global é adicionado na aplicação ao importar o **`StoreModule`**. Como o **`@ngrx/store`** é inspirado no *Redux* [23], foi preciso configurar *Reducers*, *Actions* e *Selectors*. Mas nenhuma dessas configurações necessitou de código reativo por parte da aplicação.

Entretanto, o **`@ngrx/store`** é responsável apenas pelo gerenciamento do estado e não da execução de efeitos colaterais. Para adicionar efeitos colaterais em reação a mudanças no estado global é necessário utilizar a biblioteca **`@ngrx/effects`**, também citada na Seção 3.1.8.

A adição dos efeitos colaterais também é feita no módulo raiz (Código 4.3) através da importação de **`EffectsModule`**. Nessa aplicação foi configurado apenas um estado global, **`team`**, que representa um time de até 6 *Pokémons* que são adicionados ou removidos a partir da UI da aplicação. O Código 4.4 contém o efeito colateral **`TeamEffects`** que é utilizado na importação do **`EffectsModule`**.

Código 4.4 - team.effects.ts

```
import { Injectable } from "@angular/core";
import { MatSnackBar } from "@angular/material/snack-bar";
import { Actions, createEffect, ofType } from "@ngrx/effects";
import { Store } from "@ngrx/store";
import { pairwise, startWith, tap, withLatestFrom } from "rxjs";
import { AppState } from "../app.state";
import { addPokemon } from './team.actions';
import { selectTeam} from './team.selectors';

@Injectable()
export class TeamEffects {

 constructor(
```

```
    private actions$: Actions,
    private store: Store<AppState>,
    private snackBar: MatSnackBar
  ) {}

  warnInvalidAddition$ = createEffect(() => this.actions$.pipe(
    ofType(addPokemon),
    withLatestFrom(this.store.select(selectTeam).pipe(startWith([]),
pairwise())),
    tap(([actions , [previousTeam, currentTeam]]) => {
      const isTeamComplete = previousTeam.length === 6;
      if (isTeamComplete) {
        this.snackBar.open(
          'Your team cannot have more than 6 pokemons.',
          'Dismiss',
          { horizontalPosition: 'end', verticalPosition: 'top'}
        );
      }

    })
  ), { dispatch: false });
}
```

Fonte: Elaborado pelo autor, 2023.

O efeito colateral utiliza o *Observable* **`actions$`** que é fornecido para o sitema de injeção de dependências pelo sistema de gerenciamento de estado global da aplicação. Esse *Observable* emite as *Actions* que foram despachadas para o sistema, podendo ter vindo de qualquer lugar da aplicação. A lógica do efeito colateral é então executada quando a *Action* do tipo **`addPokemon`** for emitida.

A lógica do **`TeamEffects`** é responsável por exibir uma mensagem para o usuário quando ele tenta adicionar *Pokémon* ao time após ter atingido a capacidade máxima de 6 *Pokémon* do time.

4.3.1 Testes

Os seguintes testes foram desenvolvidos utilizando as 3 abordagens:

- *it should open a snackbar if attempts to create a team with more than 6 pokemon*
- *it should not open a snackbar if attempts to create a team within the size limit*

Comparações a respeito dos testes, mostrando trechos de código, são feitas na Seção 4.5.

4.4 *Pokedex Component*

Esse é o único componente da aplicação que contém alguma lógica, sendo os demais componentes apenas apresentacionais. Ele consegue utilizar métodos do *Pokedex Store* (Seção 4.2) para atualizar o estado local. Esse componente também consegue despachar *Actions* para o sistema de gerenciamento de estado global para atualizar o estado global. Como consequência da alteração desses estados, efeitos colaterais podem ser executados em respostas a essas modificações.

Além de conseguir atualizar estados através dos mecanismos disponibilizados pelos serviços de gerenciamento de estados, esse componente também é capaz de ler os estados disponibilizados por esses serviços em forma de *Observable*. Sempre que os estados forem atualizados, esses *Observables* vão emitir o novo valor e a UI da aplicação irá reagir a essas mudanças automaticamente.

Os códigos 4.5 e 4.6 contém o modelo (arquivo *Typescript*) e a visão (arquivo *HTML*) respectivamente desse componente.

Código 4.5 - pokedex.component.ts

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { MatIconModule } from '@angular/material/icon';
import { MatInputModule } from '@angular/material/input';
import { MatFormFieldModule } from '@angular/material/form-field';
import { GridViewComponent } from './ui/grid-view/grid-view.component';
import { PokedexStore } from './pokedex.store';
import { FormControl, ReactiveFormsModule } from '@angular/forms';
import { combineLatest, debounceTime, distinctUntilChanged, filter,
map, Observable, startWith } from 'rxjs';
import { Store } from '@ngrx/store';
import { addPokemon, removePokemon } from
'../shared/state/team/team.actions';
import { AppState } from '../shared/state/app.state';
import { Pokemon } from '../shared/models/pokemon.model';
import { selectTeam } from '../shared/state/team/team.selectors';
import { LoaderComponent } from '../shared/ui/loader/loader.component';
import { PokedexErrorMessageComponent } from
'./ui/pokedex-error-message/pokedex-error-message.component';

interface ViewModel {
 lastPage: boolean;
 pokemonList: Pokemon[];
 hasLastRequestFailed: boolean;
```

```
 isLoading: boolean;
 team: Pokemon[];
}

@Component({
 selector: 'app-pokedex',
 standalone: true,
 imports: [
   CommonModule,
   MatIconModule,
   ReactiveFormsModule,
   MatInputModule,
   MatFormFieldModule,
   GridViewComponent,
   LoaderComponent,
   PokedexErrorMessageComponent
 ],
 templateUrl: './pokedex.component.html',
 styleUrls: ['./pokedex.component.scss'],
 providers: [PokedexStore]
})
export class PokedexComponent {
 private lastPage$ = this.pokedexStore.lastPage$;
 private pokemonList$ = this.pokedexStore.pokemonList$;
 private isLoading$ =
this.pokedexStore.requestStatus$.pipe(map(requestStatus =>
requestStatus === 'processing'));
 private hasLastRequestFailed$ = this.pokedexStore.requestStatus$.pipe(
   filter(requestStatus => requestStatus === 'success' || requestStatus
=== 'error'),
   map(requestStatus => requestStatus === 'error')
 );
 private team$: Observable<Pokemon[]>;

 protected searchControl = new FormControl('', { nonNullable: true });
 protected vm$: Observable<ViewModel>;

 constructor(private readonly pokedexStore: PokedexStore, private
readonly store: Store<AppState>) {
   const searchUpdateTrigger$ = this.searchControl.valueChanges.pipe(
     distinctUntilChanged(),
     debounceTime(500)
   );
```

```
    this.pokedexStore.searchPokemon(searchUpdateTrigger$);
    this.team$ = this.store.select(selectTeam);
    this.vm$ = combineLatest({
      isLoading: this.isLoading$.pipe(startWith(false)),
      hasLastRequestFailed:
this.hasLastRequestFailed$.pipe(startWith(false)),
      lastPage: this.lastPage$.pipe(startWith(false)),
      pokemonList: this.pokemonList$.pipe(startWith([])),
      team: this.team$.pipe(startWith([]))
    });
  }

  protected onLoadMorePokemon(): void {
    this.pokedexStore.loadNextPage();
  }

  protected onAddPokemonToTeam(pokemon: Pokemon): void {
    this.store.dispatch(addPokemon({ pokemon}));
  }

  protected onRemovePokemonFromTeam(pokemon: Pokemon): void {
    this.store.dispatch(removePokemon({ id: pokemon.id }));
  }

  protected onTryAgain(): void {
    this.pokedexStore.retryLastRequest();
  }

}
```

Fonte: Elaborado pelo autor, 2023.

Código 4.6 - pokedex.component.html

```
<div class="page-section-container" *ngIf="vm$ | async as vm">
  <app-loader *ngIf="vm.isLoading"></app-loader>

  <ng-container *ngIf="!vm.isLoading && !vm.hasLastRequestFailed">
    <mat-form-field class="search-bar" appearance="fill">
      <input matInput [formControl]="searchControl" placeholder="Search
pokemon..." value="Sushi">
    </mat-form-field>
    <app-grid-view
    [pokemonList]="vm.pokemonList"
    [loadMoreVisible]="!vm.lastPage"
    [team]="vm.team"
```

```
    (loadMore)="onLoadMorePokemon()"
    (addPokemonToTeam)="onAddPokemonToTeam($event)"
    (removePokemonFromTeam)="onRemovePokemonFromTeam($event)"
   ></app-grid-view>
 </ng-container>

 <app-pokedex-error-message *ngIf="!vm.isLoading &&
vm.hasLastRequestFailed" (tryAgain)="onTryAgain()">
 </app-pokedex-error-message>
</div>
```

Fonte: Elaborado pelo autor, 2023.

No Código 4.5 o componente define **`searchUpdateTrigger$`** que nada mais é do que um *Observable*. Ele emite valores toda vez que se passam 500 milissegundos após alguma alteração no conteúdo da barra de pesquisa. Esse *Observable* é utilizado para atualizar o estado da *Pokedex Store*, e reagindo a essa alteração é esperado que o efeito colateral faça uma requisição *HTTP* ao servidor usando a cadeira de caracteres emitida para filtrar *Pokémon*.

O componente também consegue mudar o estado de outras formas, como na função **`onLoadMorePokemon`** que é chamada quando o componente filho emite o evento **`loadMore`** (Código 4.6). Ao mudar o estado da *Pokedex Store* dessa maneira é esperado que o efeito colateral faça uma requisição *HTTP* ao servidor para trazer o resultado da próxima página de *Pokémon*.

O componente também consegue interagir com o estado global, despachando *Actions* do tipo **`addPokemon`** e **`removePokemon`** através dos métodos **`onAddPokemonToTeam`** e **`onRemovePokemonFromTeam`** respectivamente. Ambos os métodos também estão associados com eventos emitidos pelo componente filho.

Além de interagir ativamente atualizando os estados da aplicação, o componente também lê os estados relevantes para a sua necessidade. Esses estados são então combinados em um único *Observable* **`vm$`** e o *template* do componente (Código 4.6) consome o *Observable* **`vm$`**, logo, sempre que um estado de interesse é alterado essa mudança é propagada para **`vm$`** que resulta em uma atualização automática dos elementos que são renderizados por esse componente.

4.4.1 Testes

Os seguintes testes foram desenvolvidos utilizando as 3 abordagens:

- *it should only trigger search if new text is different from last one*
- *it should only trigger search when values have at least a 500ms time span*

- *it should dispatch addPokemon action when addPokemonToTeam event emits*
- *it should dispatch removePokemon action when removePokemonFromTeam event emits*

Comparações a respeito dos testes, mostrando trechos de código, são feitas na Seção 4.5.

4.5 Comparações

Nos códigos referentes aos testes utilizando *subscribing strategy*, o trecho representado pelo Código 4.7 é utilizado.

Código 4.7 - Trecho comum aos códigos que utilizam *subscribing strategy*

```
afterEach(() => {
   subscription?.unsubscribe();
 });
```

Fonte: Elaborado pelo autor, 2023.

Esse trecho de código é necessário pois os *Observables* causam problemas de vazamento de memória caso eles não completem ou a inscrição seja cancelada. Em um projeto suficientemente grande com uma enorme quantidade de testes, se esta prática não for adotada, é possível que a execução de testes consuma muito mais memória do que necessária. Em um time grande, com pressão para realizar uma entrega é um detalhe que pode passar despercebido na etapa de revisão de código e causar um problema ou gastos desnecessários a longo prazo.

Existem cenários onde *subscribing strategy* é forçado a quebrar o padrão Organizar (*Arrange*), Agir (*Act*) e Verificar (*Assert*) [30], já bem definido na industria, o que prejudica a legibilidade do teste. Esse problema foi inicialmente detectado no teste "*it should be able to retry last request using same parameters, but incrementing the retry count by 1*" (Código 4.8) onde é preciso fazer uma inscrição no *Observable* **`apiTrigger$`** antes da etapa de Agir.

Código 4.8 - should be able to retry last request using same parameters, but incrementing the retry count by 1 (*subscribing strategy)*

```
it('should be able to retry last request using same parameters, but
incrementing the retry count by 1', () => {
   service = TestBed.inject(PokedexStore);
   const emittedValues: any[] = [];
   const apiTrigger$ = service.select(state => state.apiTrigger);
   subscription = apiTrigger$.subscribe(state => {
     emittedValues.push(state);
   });

   service.searchPokemon('test1');
```

```
   pokeApiServiceSpy.getPokemonList.and.returnValue(of({ last: true,
content: secondPageOfPokemonData }));
   service.loadNextPage();
   service.retryLastRequest();

   const currentValue = emittedValues.slice(-1)[0];
   const previousValue = emittedValues.slice(-2,-1)[0];
   expect(currentValue).toEqual({
     ...previousValue,
     requestRetryCount: previousValue.requestRetryCount + 1
   });
 });
```

Fonte: Elaborado pelo autor, 2023.

As verificações estavam sendo feitas no *callback* do método *subscribe* fazendo com que a etapa de Verificar estivesse fora de ordem. Para contornar o problema, foi necessário utilizar o *callback* para preencher um vetor com os valores emitidos pelo *Observable* durante o teste. Por fim, os valores no vetor eram utilizados para fazer as verificações. Ainda que essa solução tenha sido capaz de resolver o problema da ordem das etapas do teste, ela também tem um custo na legibilidade. Essa mesma solução foi aplicada em outros testes como *"it should only trigger search if new text is different from last one"* e *"it should only trigger search when values have at least a 500ms time span*".

Os testes feitos com *observer-spy strategy* no geral foram bem semelhantes ao testes do *subscribing strategy*, porém sem os pontos negativos citados acima. A biblioteca **`observer-spy`** tira do desenvolvedor a responsabilidade de gerenciar as inscrições dos *Observables* [13], evitando a necessidade de repetir o Código 4.7 nos arquivos de teste e eliminando o risco de vazamento de memória causado pela possível falha no cancelamento de inscrições. Para que isso ocorra, basta configurar uma única vez durante a vida do projeto, que o método **`autoUnsubscribe`** seja executado no arquivo de configuração de testes [13]. No *Angular* esse é o arquivo *test.ts* mostrado pelo Código 4.9.

Código 4.9 - test.ts

```
import 'zone.js/testing';
import { getTestBed } from "@angular/core/testing";
import {
 BrowserDynamicTestingModule,
 platformBrowserDynamicTesting
} from '@angular/platform-browser-dynamic/testing';
```

```
import { autoUnsubscribe } from '@hirez_io/observer-spy';

getTestBed().initTestEnvironment(BrowserDynamicTestingModule,
platformBrowserDynamicTesting(), {
 errorOnUnknownElements: true,
 errorOnUnknownProperties: true
});

autoUnsubscribe();
```

Fonte: Elaborado pelo autor, 2023.

Os testes utilizando *observer-spy strategy* também não causam problemas na estrutura Organizar, Agir e Verificar. Tomando como exemplo o teste "*it should only trigger search if new text is different from last one"* do Código 4.10 e comparando com o mesmo teste no Código 4.11 fica perceptível que o código utilizando *observer-spy strategy* não impacta a estrutura do teste.

Código 4.10 - should only trigger search if new text is different from last one (*observer-spy strategy)*

```
it('should only trigger search if new text is different from last one',
fakeAsync(() => {
   const searchTriggerSpy = subscribeSpyTo(searchTrigger$);

   typeInSearch('test1');
   tick(500);
   typeInSearch('test1');
   tick(500);
   typeInSearch('test2');
   tick(500);

   expect(searchTriggerSpy.getValues().length).toBe(2);
   expect(searchTriggerSpy.getValueAt(0)).toBe('test1');
   expect(searchTriggerSpy.getValueAt(1)).toBe('test2');
 }));
```

Fonte: Elaborado pelo autor, 2023.

Código 4.11 - should only trigger search if new text is different from last one (*subscribing strategy)*

```
it('should only trigger search if new text is different from last one',
fakeAsync(() => {
   const emittedValues: string[] = [];
   subscription = searchTrigger$.subscribe(text =>
emittedValues.push(text));

   typeInSearch('test1');
```

```
  tick(500);
  typeInSearch('test1');
  tick(500);
  typeInSearch('test2');
  tick(500);

  expect(emittedValues.length).toBe(2);
  expect(emittedValues[0]).toBe('test1');
  expect(emittedValues[1]).toBe('test2');
}));
```

Fonte: Elaborado pelo autor, 2023.

O método **`subscribeSpyTo`** inscreve-se automaticamente no *Observable* de interesse e retorna um *Spy* que possui métodos que facilitam verificações feitas com valores emitidos pelo *Observable*. Ao fim do teste a inscrição é automaticamente cancelada.

Nos testes que utilizam *marble strategy*, os diagramas de *marble* ajudam, por utilizar uma abordagem mais visual para representar os fluxos de dados dos *Observables*. Entretanto, é necessário aprender uma sintaxe específica, complexa e bastante verbosa [12] para poder ter essa representação visual, o que compromete a legibilidade do teste.

Esse problema de legibilidade da *marble strategy* fica bastante nítido ao comparar os testes gerados pelas 3 ferramentas. Um bom exemplo é o teste *"it should be able to fetch data by search text"* que nos códigos 4.12 e 4.13 são bem mais simples do que no Código 4.14.

Código 4.12 - it should be able to fetch data by search text (*subscribing strategy)*

```
it('should be able to fetch data by search text', () => {
  service = TestBed.inject(PokedexStore);
  const state$ = service.select(state => state);
  const searchStream$ = of('test1','test2','test3');

  service.searchPokemon(searchStream$);

  subscription = state$.subscribe(state => {
    expect(state.apiTrigger.currentPage).toBe(0);
    expect(state.apiTrigger.searchText).toBe('test3');
  });
  expect(pokeApiServiceSpy.getPokemonList).toHaveBeenCalledTimes(4);
  expect(pokeApiServiceSpy.getPokemonList).toHaveBeenCalledWith(0,
'test1');
```

```
   expect(pokeApiServiceSpy.getPokemonList).toHaveBeenCalledWith(0,
'test2');
   expect(pokeApiServiceSpy.getPokemonList).toHaveBeenCalledWith(0,
'test3');
 });
```

Fonte: Elaborado pelo autor, 2023.

Código 4.13 - it should be able to fetch data by search text (*observer-spy strategy)*

```
it('should be able to fetch data by search text', () => {
   service = TestBed.inject(PokedexStore);
   const searchStream$ = of('test1','test2','test3');
   const stateSpy = subscribeSpyTo(service.select(state => state));

   service.searchPokemon(searchStream$);

   expect(stateSpy.getLastValue()?.apiTrigger.currentPage).toEqual(0);

expect(stateSpy.getLastValue()?.apiTrigger.searchText).toEqual('test3')
;
   expect(pokeApiServiceSpy.getPokemonList).toHaveBeenCalledTimes(4);
   expect(pokeApiServiceSpy.getPokemonList).toHaveBeenCalledWith(0,
'test1');
   expect(pokeApiServiceSpy.getPokemonList).toHaveBeenCalledWith(0,
'test2');
   expect(pokeApiServiceSpy.getPokemonList).toHaveBeenCalledWith(0,
'test3');
 });
```

Fonte: Elaborado pelo autor, 2023.

Código 4.14 - it should be able to fetch data by search text (*marble strategy)*

```
it('should be able to fetch data by search text', () => {
   const requestMarble = '-----a|';
   const searchMarble =  '----------a--------b';
   //                     -----x    -----x   -----x (requests)
   const resultMarble =  'a----b----(cd)-e---(fg)-h';

   pokeApiServiceSpy.getPokemonList.and.returnValue(cold(requestMarble,
{ a: { last: false, content: firstPageOfPokemonData} }));
   service = TestBed.inject(PokedexStore);

   const state$ = service.select(state => state);
   const searchStream$ = cold(searchMarble, { a: 'test1', b: 'test2'
});
   const expectedState$ = cold(resultMarble, {
```

```
      a: { pokemonList: [], requestStatus: 'processing', apiTrigger: {
requestRetryCount: 0, currentPage: 0, lastPage: false, searchText: ''
}},
      b: { pokemonList: firstPageOfPokemonData, requestStatus:
'success', apiTrigger: {requestRetryCount: 0, currentPage: 0, lastPage:
false, searchText: '' }},
      c: { pokemonList: [], requestStatus: 'success', apiTrigger: {
requestRetryCount: 0, currentPage: 0, lastPage: false, searchText:
'test1' }},
      d: { pokemonList: [], requestStatus: 'processing', apiTrigger: {
requestRetryCount: 0, currentPage: 0, lastPage: false, searchText:
'test1' }},
      e: { pokemonList: firstPageOfPokemonData, requestStatus:
'success', apiTrigger: { requestRetryCount: 0, currentPage: 0,
lastPage: false, searchText: 'test1' }},
      f: { pokemonList: [], requestStatus: 'success', apiTrigger: {
requestRetryCount: 0, currentPage: 0, lastPage: false, searchText:
'test2' }},
      g: { pokemonList: [], requestStatus: 'processing', apiTrigger: {
requestRetryCount: 0, currentPage: 0, lastPage: false, searchText:
'test2' }},
      h: { pokemonList: firstPageOfPokemonData, requestStatus:
'success', apiTrigger: { requestRetryCount: 0, currentPage: 0,
lastPage: false, searchText: 'test2' }},
    });

    service.searchPokemon(searchStream$)

    expect(state$).toBeObservable(expectedState$);
    expect(pokeApiServiceSpy.getPokemonList).toHaveBeenCalledTimes(3);
    expect(pokeApiServiceSpy.getPokemonList).toHaveBeenCalledWith(0,
'test1');
    expect(pokeApiServiceSpy.getPokemonList).toHaveBeenCalledWith(0,
'test2');
  });
```

Fonte: Elaborado pelo autor, 2023.

Além de difíceis de ler, os testes feitos com *marble* também são os mais complexos de se desenvolver. Enquanto nas outras abordagens os testes precisavam focar apenas em um ou dois valores emitidos pelos *Observables*, nos testes dessa abordagem é necessário representar o fluxo de dados como um todo, ainda que não fossem relevantes ao que se estava de fato sendo testado.

Outro problema dessa abordagem aparece quando é necessário fazer testes que simulam a ação de um usuário. O teste *"it should dispatch addPokemon action when addPokemonToTeam event emits"* é consideravelmente mais simples nos códigos 4.15 e 4.16 do que no Código 4.17.

Código 4.15 - it should dispatch addPokemon action when addPokemonToTeam event emits (*subscribing strategy*)

```
 it('should dispatch addPokemon action when addPokemonToTeam event
emits', () => {
   const pokemon = createPokemonMock();
   const expectedAction = addPokemon({ pokemon})

   gridViewComponent.triggerEventHandler('addPokemonToTeam', pokemon);

   subscription = store.scannedActions$.subscribe(action => {
     expect(action).toEqual(expectedAction);
   })
 });
```

Fonte: Elaborado pelo autor, 2023.

Código 4.16 - it should dispatch addPokemon action when addPokemonToTeam event emits (*observer-spy strategy*)

```
 it('should dispatch addPokemon action when addPokemonToTeam event
emits', () => {
   const pokemon = createPokemonMock();
   const expectedAction = addPokemon({ pokemon});
   const scannedActionsSpy = subscribeSpyTo(store.scannedActions$);

   gridViewComponent.triggerEventHandler('addPokemonToTeam', pokemon);

   expect(scannedActionsSpy.getLastValue()).toEqual(expectedAction);
 });
```

Fonte: Elaborado pelo autor, 2023.

Código 4.17 - it should dispatch addPokemon action when addPokemonToTeam event emits (*marble strategy*)

```
 it('should dispatch addPokemon action when addPokemonToTeam event
emits', () => {
   const dispatchMarble = '--a';
   const expectedMarble = 'a-b';
   const pokemon = createPokemonMock();
   const expectedAction = addPokemon({ pokemon});

   scheduler.run(({ cold, expectObservable }) => {
     const expected$ = cold(expectedMarble, { a: INITIAL_ACTION, b:
expectedAction });
```

```
      subscription = cold(dispatchMarble).subscribe(() => {
        gridViewComponent.triggerEventHandler('addPokemonToTeam',
pokemon);
      });

      expectObservable(store.scannedActions$).toEqual(expected$);
    });
  });
```

Fonte: Elaborado pelo autor, 2023.

Enquanto nos primeiros basta simular a emição do evento por parte do componente filho, no teste com *marble* é necessário definir um *Observable* (**`dispatchMarble`** nesse exemplo) utilizando a syntaxe especial desse tipo de teste que deve ser inscrito para que a emissão do evento seja feita em um momento específico no tempo.

Em relação à cobertura de testes, as 3 abordagens resultaram exatamente na mesma cobertura (Figura 4.1), comprovando que os testes desenvolvidos são análogos.

Figura 4.1 - Cobertura de testes

```
> pokedex-tcc@0.0.0 test:subscribing
> ng test --watch false --code-coverage --include **/*.subscribing.spec.ts

✔ Browser application bundle generation complete.
16 04 2023 23:29:09.764:INFO [karma-server]: Karma v6.4.1 server started at http://localhost:9876/
16 04 2023 23:29:09.766:INFO [launcher]: Launching browsers Chrome with concurrency unlimited
16 04 2023 23:29:09.769:INFO [launcher]: Starting browser Chrome
16 04 2023 23:29:10.565:INFO [Chrome 110.0.0.0 (Linux x86_64)]: Connected on socket KQQWOQIwxuq4Pe6VAAAB with id 5094907
Chrome 110.0.0.0 (Linux x86_64): Executed 12 of 12 SUCCESS (0.002 secs / 0.132 secs)
TOTAL: 12 SUCCESS

============================== Coverage summary ==============================
Statements   : 79.2% ( 80/101 )
Branches     : 81.81% ( 9/11 )
Functions    : 64% ( 32/50 )
Lines        : 79.01% ( 64/81 )
==============================================================================
> pokedex-tcc@0.0.0 test:observer-spy
> ng test --watch false --code-coverage --include **/*.observer-spy.spec.ts

✔ Browser application bundle generation complete.
16 04 2023 23:30:01.430:INFO [karma-server]: Karma v6.4.1 server started at http://localhost:9876/
16 04 2023 23:30:01.432:INFO [launcher]: Launching browsers Chrome with concurrency unlimited
16 04 2023 23:30:01.436:INFO [launcher]: Starting browser Chrome
16 04 2023 23:30:02.212:INFO [Chrome 110.0.0.0 (Linux x86_64)]: Connected on socket JeXgicdggyPwyHbqAAAB with id 71717003
Chrome 110.0.0.0 (Linux x86_64): Executed 12 of 12 SUCCESS (0.001 secs / 0.13 secs)
TOTAL: 12 SUCCESS

============================== Coverage summary ==============================
Statements   : 79.2% ( 80/101 )
Branches     : 81.81% ( 9/11 )
Functions    : 64% ( 32/50 )
Lines        : 79.01% ( 64/81 )
==============================================================================
> pokedex-tcc@0.0.0 test:marble
> ng test --watch false --code-coverage --include **/*.marble.spec.ts

✔ Browser application bundle generation complete.
16 04 2023 23:29:38.520:INFO [karma-server]: Karma v6.4.1 server started at http://localhost:9876/
16 04 2023 23:29:38.523:INFO [launcher]: Launching browsers Chrome with concurrency unlimited
16 04 2023 23:29:38.527:INFO [launcher]: Starting browser Chrome
16 04 2023 23:29:39.291:INFO [Chrome 110.0.0.0 (Linux x86_64)]: Connected on socket eU_9645DRMwYl59YAAAB with id 87436182
Chrome 110.0.0.0 (Linux x86_64): Executed 12 of 12 SUCCESS (0.001 secs / 0.129 secs)
TOTAL: 12 SUCCESS

============================== Coverage summary ==============================
Statements   : 79.2% ( 80/101 )
Branches     : 81.81% ( 9/11 )
Functions    : 64% ( 32/50 )
Lines        : 79.01% ( 64/81 )
==============================================================================
```

Fonte: Elaborado pelo autor, 2023.

A quantidade de linhas de código que cada abordagem precisou para fazer os testes foi registrada na Tabela 4.1. A abordagem *observer-spy strategy* foi a que menos precisou de linhas para testar a aplicação, já a abordagem que mais precisou de linhas foi a *marble strategy*, com quase 100 linhas a mais que a primeira colocada. A abordagem *subscribing strategy*, apesar de estar na segunda colocação na comparação por quantidade de linhas, possui um número próximo ao da primeira colocada. Essa pequena quantidade de linhas a mais ocorre pois em algumas situações a *subscribing strategy* precisa utilizar um vetor para armazenar os dados e também precisa aplicar uma lógica para se desinscrever dos *Observables*.

Tabela 4.1 - Abordagem vs Linhas de código do teste

	Pokedex Store	*Global Store*	*Pokedex Component*	TOTAL
subscribing strategy	137	79	130	346
marble strategy	186	100	138	424
observer-spy strategy	136	74	120	330

Fonte: Elaborado pelo autor, 2023.

Quanto à finalidade das asserções, as abordagem *subscribing strategy* e *observer-spy strategy* são muito semelhantes, elas utilizam de um ou mais valores de emissões individuais que um *Observable* emitiu. Entretanto, a segunda ganha um pouco de vantagem por possuir métodos que facilitam na obtenção dos valores dessas emissões.

A *marble strategy* compara todos os valores emitidos por um *Observable* de uma única vez, levando em consideração não só quais valores foram emitidios mas também quando foram emitidos pois essa abordagem utiliza tempo virtual.

Por apresentar uma comparação mais completa, os testes feitos com *marble strategy* ganham a comparação a respeito da finalidade das asserções.

5. CONCLUSÃO

Esse trabalho comparou o uso de 3 ferramentas que podem ser utilizadas para realizar testes unitários em aplicações *Angular* programadas de forma reativa utilizando *RxJS*.

Para alcançar esse objetivo foi desenvolvida uma aplicação *web* com o *framework Angular*, um dos mais populares no mercado. Testes unitários análogos foram desenvolvidos para esse *software* para que fosse possível a comparação entre as ferramentas selecionadas. As métricas utilizadas para fazer essa comparação foram: cobertura de teste, quantidade de linhas, legibilidade do teste e finalidade das asserções.

A primeira ferramenta é a própria biblioteca *RxJS* que através do método *subscribe* dos *Observables* é capaz de ler os dados emitidos por eles e usa-los para fazer asserções.

A segunda ferramenta é uma *API* específica para testes disponibilizada pelo *RxJS*. Essa *API* cria e compara todo o fluxo de dados emitidos por *Observables* a partir de uma estrutura visual, os diagramas de *marble*.

A terceira ferramenta é uma biblioteca (**`observer-spy`**) criada especificamente para testar código feito com a biblioteca *RxJS*.

Apesar dos testes feitos com o diagrama de *marble* possuírem uma asserção mais completa, esse poder vem com um custo muito alto pois aumenta a complexidade dos testes, diminui drasticamente a legibilidade dos testes e necessitam de uma quantidade muito maior de código do que as outras duas abordagens por ser mais verbosa. Embora essa ferramenta possa ser útil em cenários complexos onde seja necessário comparar fluxos de dados inteiros, é importante lembrar que apenas desenvolvedores experientes com ela vão ser capazes de desenvolver ou mesmo entender tais testes, sendo preferível quebrar cenários complexos em múltiplos cenários mais simples sempre que possível.

Apesar da primeira abordagem, que utiliza a própria biblioteca do *RxJS*, ser mais legível e necessitar menos código que a ferramenta baseada em diagramas de *marble* ela perde em todos os critérios para o uso da biblioteca **`observer-spy`**, além de poder causar problemas de vazamento de memória por exigir que o desenvolvedor cancele as inscrições manualmente.

A biblioteca **`observer-spy`** se provou como sendo a melhor solução para testar esse tipo de aplicação. Ela é mais legível, necessita de menos códigos e verificações, apesar de menos completa que a abordagem *marble* é suficiente para a grande maioria dos cenários. Além disso, a própria biblioteca é capaz de gerenciar o cancelamento das inscrições dos *Observables* de forma automática, exigindo apenas uma configuração que é feita uma única vez durante a vida do projeto.

5.1 Trabalhos futuros

Durante o desenvolvimento desse trabalho o time do *Angular* abriu uma *RFC* para adotar o *Signals* [31], solução existente no *SolidJS* como uma nova primitiva para reatividade do *Angular* [32]. Ainda que esteja planejado a interoperabilidade dos *Signals* com o *RxJS*, essa nova solução deve substituir o uso do *RxJS* em cenários mais simples quando criando aplicações *Angular* com o paradigma de programação reativa, fazendo com que o poder e complexidade do *RxJS* sejam necessárias apenas para funcionalidades mais complexas.

Como sugestão para trabalhos futuros, esse trabalho recomenda a exploração de como a adoção dos *Signals* devem impactar o desenvolvimento e teste de aplicações *Angular* com o paradigma reativo.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Noleto, Cairo. Aplicações web: entenda o que são e como funcionam. **Blog da Trybe**, 2022. Disponível em: <https://blog.betrybe.com/desenvolvimento-web/aplicacoes-web>. Acesso em: 01/04/2023.

[2] NETO, Arilo; CLAUDIO, Dias. Introdução a teste de software. Engenharia de Software Magazine, v. 1, p. 22, 2007.

[3] Pittet, Sten. The different types of software testing. **Atlassian**. Disponível em: <https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing>. Acesso em: 01/04/2023.

[4] Reactive programming. **Wikipedia**. Disponível em <https://en.wikipedia.org/wiki/Reactive_programming>. Acesso em: 01/04/2023.

[5] Escoffier, Clement. 5 Things to Know About Reactive Programming. **Red Hat Developer**, 2017. Disponível em: <https://developers.redhat.com/blog/2017/06/30/5-things-to-know-about-reactive-programming>. Acesso em: 01/04/2023.

[6] What is Angular? **Angular**. Disponível em: <https://angular.io/guide/what-is-angular>. Acesso em: 01/04/2023.

[7] FRONT-END FRAMEWORKS. **State of JS**, 2022. Disponível em: <https://2022.stateofjs.com/en-US/libraries/front-end-frameworks/>. Acesso em: 01/04/2023.

[8] Introduction. **RxJS**. Disponível em: <https://rxjs.dev/guide/overview>. Acesso em: 02/04/2023.

[9] Morony, Joshua. The easier way to code Angular apps. **Youtube**, 2023. Disponível em <https://www.youtube.com/watch?v=skOTEbGwncE>. Acesso em: 02/04/2023.

[10] Pearson, Mike. 5 reasons to avoid imperative code. **DEV**, 2022. Disponível em: <https://dev.to/this-is-learning/5-reasons-to-avoid-imperative-code-e09> . Acesso em: 15/04/2023.

[11] Ferreira, Pamella. Estudo comparativo entre ferramentas de teste para React. Orientador: Leopoldo Motta Teixeira. 2021. 46 f. TCC (Graduaçã) - Ciência da

Computação, Centro de Informática, Universidade Federal de Pernambuco, Recife. 2021.

[12] Testing RxJS Code with Marble Diagrams. **RxJS**. Disponível em: <https://rxjs.dev/guide/testing/marble-testing>. Acesso em: 02/04/2023.

[13] @hirez_io/observer-spy. **GitHub**. Disponível em: <https://github.com/hirezio/observer-spy>. Acesso em: 02/04/2023.

[14] Angular components overview. **Angular**. Disponível em: <https://angular.io/guide/component-overview>. Acesso em: 08/04/2023.

[15] Creating an injectable service. **Angular**. Disponível em: <https://angular.io/guide/creating-injectable-service>. Acesso em: 08/04/2023.

[16] CLI Overview and Command Reference. **Angular**. Disponível em: <https://angular.io/cli>. Acesso em: 08/04/2023.

[17] Basics of testing components. **Angular**. Disponível em: <https://angular.io/guide/testing-components-basics>. Acesso em: 09/04/2023.

[18] Testing services. **Angular**. Disponível em: <https://angular.io/guide/testing-services>. Acesso em: 09/04/2023.

[19] What is a Test Runner. **BrowserStack**. Disponível em: <https://www.browserstack.com/guide/what-is-test-runner>. Acesso em: 08/04/2023.

[20] Testing. **Angular**. Disponível em: <https://angular.io/guide/testing>. Acesso em: 08/04/2023.

[21] Imperative programming. **Wikipedia**. Disponível em: <https://en.wikipedia.org/wiki/Imperative_programming>.Acesso em: 08/04/2023.

[22] Declarative Programming. **Wikipedia**. Disponível em: <https://en.wikipedia.org/wiki/Declarative_programming>. Acesso em: 08/04/2023.

[23] What is NgRx? **NgRx**. Disponível em: <https://ngrx.io/docs>. Acesso em: 08/04/2023.

[24] Fat, Nina; Vujovic, Marijana; Papp, Istvan; Novak, Sebastian. "Comparison of AngularJS framework testing tools." 2016 Zooming Innovation in Consumer Electronics International Conference (ZINC). IEEE, 2016. Disponível em: <https://ieeexplore.ieee.org/document/7513659>. Acesso em: 09/04/2023.

[25] *API* v2. **PokeAPI**. Disponível em: <https://pokeapi.co/docs/v2>. Acesso em: 15/04/2023.

[26] Figueirôa, André. Pokedex TCC Backend. **Github**, 2023. Disponível em: <https://github.com/FigueiroaAndre/pokedex-tcc-backend>. Acesso em: 15/04/2023.

[27] Figueirôa, André. PokedexTcc. **Github**, 2023. Disponível em: <https://github.com/FigueiroaAndre/pokedex-tcc>. Acesso em: 15/04/2023.

[28] Material Design components for Angular. **Angular Material**. Disponível em: <https://material.angular.io/>. Acesso em: 15/04/2023.

[29] Testing HTTP requests. **Angular**. Disponível em: <https://angular.io/guide/http#testing-http-requests>. Acesso em: 16/04/2023.

[30] Noções básicas de teste de unidade. **Microsoft**, 2022. Disponível em: <https://learn.microsoft.com/pt-br/visualstudio/test/unit-test-basics?view=vs-2022>. Acesso em: 16/04/2023.

[31] Introduction/Sinals. **SolidJS**. Disponível em: <https://www.solidjs.com/tutorial/introduction_signals>. Acesso em: 17/04/2023.

[32] Rickabaugh, Alex; Scott, Andrew; Hunn, Dylan; Melbourne, Jeremy; Kozlowski, Pawel. RFC: Angular Signals. **Github**, 2023. Disponível em: <https://github.com/angular/angular/discussions/49685>. Acesso em: 17/04/2023.