# TDC 2021

## Reactive Programming

August . 2021

# Index

- Manifesto Reativo
- Programação Reativa
- Como começar
- Quem Apoia e Usa
- Boas Práticas

# Manifesto Reativo

## Sistemas Reativos

Sistemas Reativos devem ser consistente em relação a arquitetura de sistemas, e os aspectos necessários já são reconhecidos individualmente: Responsivos, Resilientes, Elásticos e Orientados a Mensagens.

# Responsivos

O sistema responde em um tempo razoável. Esse comportamento consistente, por sua vez, simplifica tratamentos de erro, reforça a confiança do usuário final e incentiva interações futuras.

C.E.S.A.R

# Resilientes

Resiliência é obtida por meio de replicação, contenção, isolamento e delegação. Falhas são contidas dentro de cada componente, isolando-os uns dos outros.

C.E.S.A.R

# Elásticos

Reagir a mudanças na taxa de solicitações por meio do aumento ou diminuição dos recursos alocados para lidar com essas entradas.

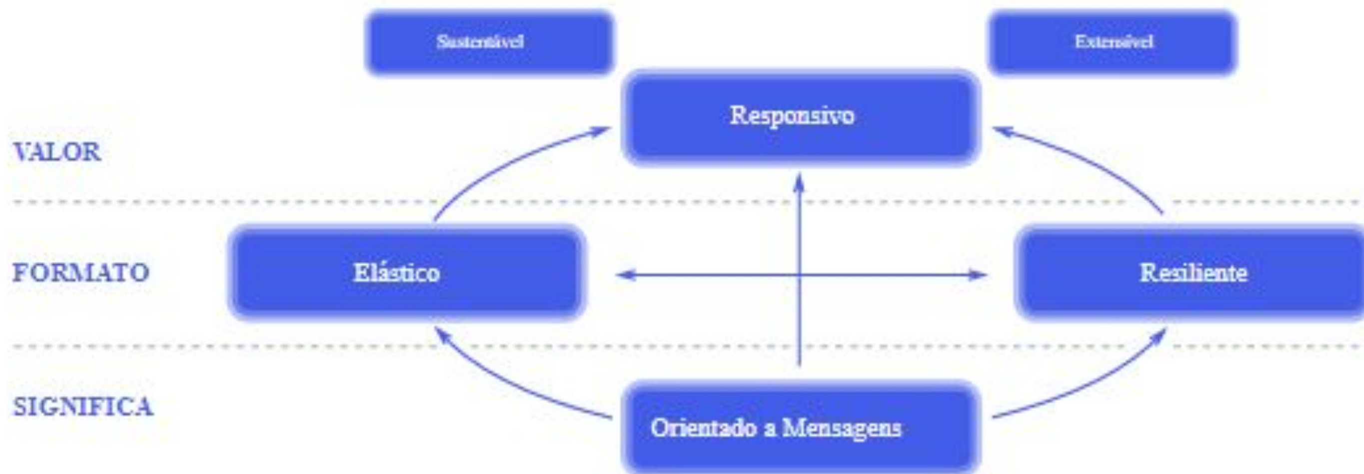
C.E.S.A.R

# Orientados a Mensagens

Baseiam-se em transferência de mensagens assíncronas para estabelecer fronteiras entre os componentes e garantir baixo acoplamento, isolamento, transparência na localização

C.E.S.A.R

# Sistemas Reativos



# O que é Programação Reativa ?

Programação reativa no contexto de desenvolvimento de software foi citado pela primeira vez por Gérard Berry no artigo “Real time programming : special purpose or general purpose languages”.

Everything is a stream

É um paradigma de programação orientado a fluxo de dados e propagação de mudança.

# Como começar

# ReactiveX

# Escolha de acordo com o que você usa



## Languages

- Java: RxJava
- JavaScript: RxJS
- C#: Rx.NET
- C#(Unity): UniRx
- Scala: RxScala
- Clojure: RxClojure
- C++: RxCpp
- Lua: RxLua
- Ruby: Rx.rb
- Python: RxPY
- Go: RxGo
- Groovy: RxGroovy
- JRuby: RxJRuby
- Kotlin: RxKotlin
- Swift: RxSwift
- PHP: RxPHP
- Elixir: reaxive
- Dart: RxDart

## ReactiveX for platforms and frameworks

- RxNetty
- RxAndroid
- RxCocoa

# Quem apoia ?

# Quem utiliza ?

C.E.S.A.R

NETFLIX

GitHub

futurice

O.C.TANNER

# BOAS PRÁTICAS

# Evite a lógica dentro de “subscribe”

Lógica dentro do subscribe

```
1  pokemon$.subscribe((pokemon: Pokemon) => {
2    if (pokemon.type === "Water") {
3      return;
4    }
5    const pokemonStats = getStats(pokemon);
6    logStats(pokemonStats);
7    saveToPokedex(pokemonStats);
8  });
```

Utilização de pipeable operators

```
1  pokemon$
2    .pipe(
3      filter(({ type }) => type !== "Water"),
4      map(pokemon => getStats(pokemon)),
5      tap(stats => logStats(stats))
6    )
7    .subscribe(stats => saveToPokedex(stats));
```

C.E.S.A.R

## Force a conclusão dos Observables

### Manual unsubscribing

```
1  pokemonSubscription = pokemon$.subscribe(pokemon => {
2    // Do something with pokemon
3  });
4
5  pokemonSubscription.unsubscribe();
```

### Utilize ‘takeUntil’ para concluir

```
13  number$
14    .pipe(takeUntil(stop$)).subscribe(number => {
15      // Do something with number
16    });
17
18  function stop() {
19    stop$.next();
20    stop$.complete();
21  }
```

C.E.S.A.R

# Evite Lógica Duplicada

## Filter duplicado nos subscribes

```
7   number$
8     .pipe(
9       filter<number>(Boolean),
10      reduce((acc, curr) => acc + curr)
11    )
12    .subscribe(n => console.log(`Total: ${n}`));
13
14  // Emit even numbers
15  number$
16    .pipe(
17      filter<number>(Boolean),
18      filter(n => n % 2 === 0)
19    )
20    .subscribe(console.log);
```

## Utilização de pipe na stream

```
4   const number$ = from([null, 2, 1, 0, 5, false, 6, 7]).pipe(
5     filter<number>(Boolean)
6   );
7
8   // Adding numbers
9   number$
10    .pipe(
11      reduce((acc, curr) => acc + curr)
12    )
13    .subscribe(n => console.log(`Total: ${n}`));
```

# Evite subscribe aninhados

C.E.S.A.R

## Subscribe aninhados

```
1  getTrainer().subscribe(trainer =>
2    getStarterPokemon(trainer).subscribe(pokemon =>
3      // Do stuff with pokemon
4    )
5  );
```

## Utilize funções de mapping

```
1  getTrainer()
2    .pipe(
3      switchMap(trainer => getStarterPokemon(trainer))
4    )
5    .subscribe(pokemon => {
6      // Do stuff with pokemon
7    });
```

Não passe streams para componentes diretamente

```
// BAD
// app.component.ts
@Component({
    selector: 'app',
    template: `
        <!--
            BAD: The users$ steram is passed
            to user-detail directly as a stream
        -->
        <user-detail [user$]="user$"></user-detail>
    `
})
class AppComponent {
    // this http call will get called when the
    // user-detail component subscribes to users$
    // We don't want that
    users$ = this.http.get(...);
    ...
}
```

C.E.S.A.R

# Não passe streams para serviços

C.E.S.A.R

```
// BAD
// app.component.ts
class AppComponent {
     users$ = this.http.get(...)
     filteredusers$ = this.fooService
        .filterUsers(this.users$); // Passing stream directly: BAD
    ...
}

// foo.service.ts
class FooService {
    // return a stream based on a stream
    // BAD! because we don't know what will happen here
    filterUsers(users$: Observable<User[]>): Observable<User[]> {
        return users$.pipe(
            map(users => users.filter(user => user.age >= 18))
    }
}
```

# Clean Code

- Alinhe os operadores uns abaixo dos outros
- Extraia em fluxos diferentes quando se tornar ilegível
- Coloque funcionalidades mais complexas em métodos privados

```
foo$.pipe(
    map(...)
    filter(...)
    tap(...)
)
```

C.E.S.A.R

## Gabriel Silva

- Formado em Ciência da Computação.
- 4 Anos de experiência (Web, Android, Embarcados).
- Amo aprender e compartilhar conhecimento.

TDC 2021

# That's all Folks!

August/ 2021