

Angular

- Guard
- Observables
- Subjects
- Comunicação entre componentes não relacionados

Routes Guards

- Utilizado para controlar se o usuário pode navegar para ou fora da rota atual;
- Permitir que o usuário navegue por todas as partes do aplicativo não é uma boa ideia;
- É preciso restringir o usuário até que ele execute ações específicas, como o login;
- Angular fornece os **Route Guards** para essa finalidade.

Onde usar?

- Um dos cenários comuns em que são usados os guards de rotas é a autenticação;
- Faz com que o aplicativo impeça que o usuário não autorizado acesse a rota protegida;
- Esse impedimento é alcançado usando o guarda CanActivate, que o angular invoca quando o usuário tenta navegar na rota protegida;
- Em seguida, o guarda é conectado e é utilizado o serviço de autenticação para verificar se o usuário está autorizado a usar a rota ou não;
- Caso o usuário não esteja autorizado, é possível redirecionar o usuário para a página de login.

rotasApp: Routes

```
const rotasApp: Routes = [  
  { path: '', canActivate:[guardaGuard], canDeactivate:[guardaDesativarGuard], component: HomeComponent},  
  { path: 'login',component: LoginComponent },  
  {path: 'usuarios', canActivate:[guardaGuard], component: UsuariosComponent},  
  { path: '**', redirectTo: '' }  
];
```

Tipos de guardas de rota

- ng g g Guarda

```
❖ > ng g g Guarda
? Which type of guard would you like to create? (Press <space> to select, <a> to toggle all,
<i> to invert selection,
and <enter> to proceed)
>(*) CanActivate
( ) CanActivateChild
( ) CanDeactivate
( ) CanMatch
```

CanActivate

- Decide se uma rota pode ser ativada;
- Essa proteção é útil nas circunstâncias em que o usuário não está autorizado a navegar até o componente de destino;
- O usuário pode não estar logado no sistema

CanActivate

```
import { CanActivateFn, Router } from '@angular/router';
import { AutenticacaoService } from '../autenticacao.service';
import { inject } from '@angular/core';

export const guardaGuard: CanActivateFn = (route, state) => {
  const servicoAutenticacao = inject(AutenticacaoService);
  const router = inject(Router);

  if(servicoAutenticacao.isAutenticado()) {
    console.log('autenticado');
    return true;
  }else{
    console.log('Não autenticado');
    router.navigate(['/login']);
    return false;
  }
};
```


CanDeactivate

- Decide se o usuário pode sair do componente, ou seja, pode navegar para fora da rota atual;
- É útil onde o usuário pode ter algumas alterações pendentes, que não foram salvas;
- A rota CanDeactivate permite pedir a confirmação do usuário antes de sair do componente;
- É possível perguntar ao usuário se ele quer se desautenticar do sistema pois existem alterações pendentes sem salvar.

CanDeactivate

```
import { CanDeactivateFn, Router } from '@angular/router';
import { AutenticacaoService } from '../autenticacao.service';
import { inject } from '@angular/core';

export const guardaDesativarGuard: CanDeactivateFn<unknown> = (
  component,
  currentRoute,
  currentState,
  nextState) => {

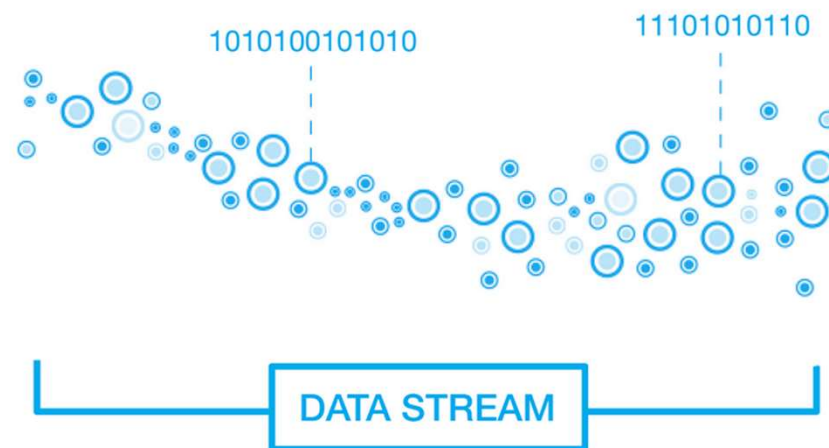
  const servicoAutenticacao = inject(AutenticacaoService);
  const router = inject(Router);
  if(servicoAutenticacao.isAutenticado()) {
    console.log('autenticado');
    const escolhaDoUsuario = confirm('Deseja sair da pagina?');
    if(escolhaDoUsuario){
      return true;
    }else {
      return false;
    }
  }
  return false;
}
```

Programação Reativa (RxJs)

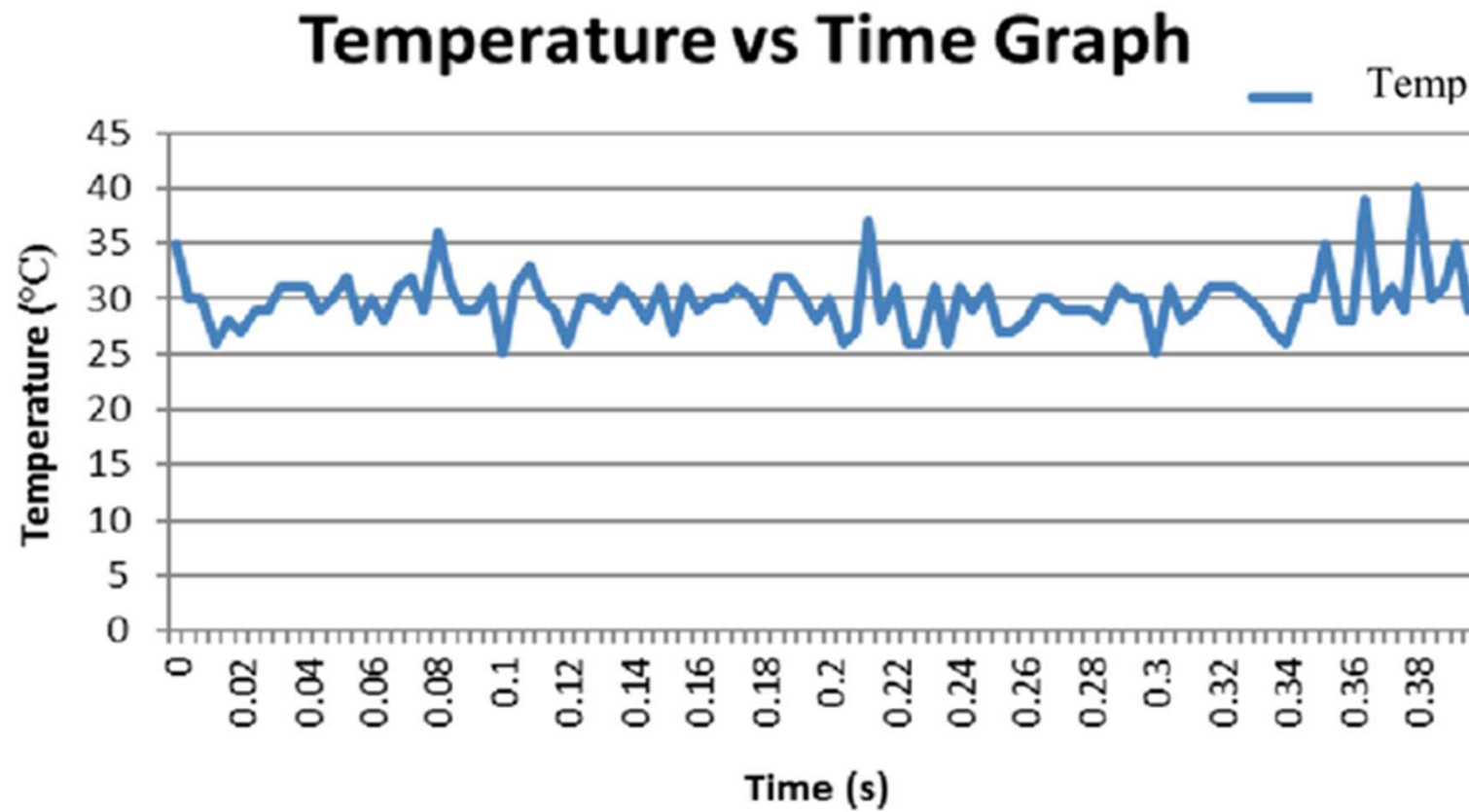
- Rx significa programação reativa;
- A programação reativa é a programação com fluxos de dados assíncronos;
- O que é um fluxo de dados?

Fluxo de dados (Data Stream)

- São dados que chegam ao longo de algum tempo;
- Pode ser qualquer coisa: variáveis, entradas do usuário, propriedades, caches, estruturas de dados e até falhas, etc.



Fluxo de dados (Data Stream)



Fluxo de dados (Data Stream)

- O fluxo de dados pode ser qualquer coisa:
 - Medições de sensores (Temperatura, tremores terreno, umidade do solo, etc);
 - Eventos de clique de mouse;
 - Notificações de usuário;
 - Eventos de teclado (Aperta a tecla, solta a tecla, etc);
 - Dados de servidores da Web depois de uma solicitação HTTP;
 - Feed do Twitter
- Observações:
 - Emitir 1 ou mais valores em qualquer momento;
 - Pode ocorrer erros;
 - Deve emitir um sinal completo quando for concluído (no caso de fluxos finitos);
 - Podem emitir dados infinitos e nunca terminar.

Fluxo de dados (Data Stream)

- Um fluxo pode ser usado como entrada para outro fluxo;
- Até mesmo vários fluxos podem ser usados como entradas para outro fluxo;
- É possível mesclar dois fluxos: filtrar um fluxo para obter outro que tenha apenas os eventos de um interesse específico;
- Possibilidade de mapear valores de dados de um fluxo para outro novo.

Design Observer

- Esses eventos emitidos são capturados apenas de forma assíncrona:
 - Definir uma função que será executada quando um valor for emitido (next);
 - Definir outra função quando um erro for emitido (error);
 - Por fim, definir uma função quando 'concluído' for emitido (complete).
- Algumas vezes as duas últimas funções (erro + conclusão) podem ser omitidas para enfatizar apenas na definição da função para valores.
- A “escuta” do stream é chamada de assinatura e as funções definidas são observadores.
- O fluxo é o sujeito (ou "observável") que está sendo observado. Este é precisamente o Padrão de Design Observer.

Observable vs Promises

- Uma Promise é simplesmente um Observable com um único valor emitido. Os fluxos Rx vão além das Promises, permitindo muitos valores retornados;
- Os Observables são ideais para lidar com fluxos de dados, como entrada do usuário, eventos WebSocket ou respostas HTTP, onde vários valores podem ser emitidos ao longo do tempo.
- As Promises são adequadas para operações assíncronas únicas, como a busca de dados de uma API.

Observable

```
getNumeros(){  
  this.valService.getNumeros().subscribe(numero => {  
    console.log(numero);  
  })  
}
```

Aula10\1.1_observable\src\app\home\home.component.ts

```
getNumeros():Observable<any>{  
  const source = from([1, 2, 3, 4, 5]);  
  return source;  
}
```

Aula10\1.1_observable\src\app\services\exemplos-observable.service.ts

Observable

```
getValores():Observable<any>{  
  const observable = new Observable((subscriber) => {  
    subscriber.next(1);  
    subscriber.next(2);  
    subscriber.next(3);  
    setTimeout(() => {  
      subscriber.next(4);  
      subscriber.complete();  
    }, 1000);  
  });  
  return observable;  
}
```

Aula10\1.1_observable\src\app\services\exemplos-observable.service.ts

```
getValores(){  
  this.valService.getValores().subscribe(valor => {  
    this.valores.push(valor);  
  })  
}
```

Aula10\1.1_observable\src\app\home\home.component.ts

RxJS pipe()

- O pipe() do RxJS é uma função autônoma e um método na interface Observable que pode ser usado para combinar vários operadores RxJS para compor operações assíncronas;
- Usa um ou mais operadores e retorna um RxJS Observable.

```
getNumeros(){  
  //retorna um Observable  
  const fonte = this.valService.getNumeros();  
  //adiciona 100 a cada número  
  const exemplo = fonte.pipe(  
    map(val => val + 100),  
  );  
  const inscricao = exemplo.subscribe(  
    numero => console.log(numero)  
  );  
  // 101, 102, 103, 104, 105  
}
```

RxJS Operators

- Map: transforma cada valor emitido pela fonte observável e cria um novo observável após manipular cada item do observável de origem;
- debounceTime: cria um atraso para determinado intervalo de tempo e sem outra emissão de fonte;
- Filter: Operador que filtra os itens emitidos pela fonte Observável usando uma condição (predicado). Ele emite apenas os valores que satisfazem a condição e ignora o resto.
- mergeMap: mergeMap é um operador que mapeia cada valor emitido por um observável para um novo observável e, em seguida, nivela esses observáveis em um único observável, útil para cenários em que se deseja combinar vários fluxos de dados em um único fluxo e não se importa com a ordem em que eles chegam.

Map

```
getNumeros():Observable<any>{  
  const source = from([1, 2, 3, 4, 5]);  
  return source;  
}
```

Aula10\1.1_observable\src\app\services\exemplos-observable.service.ts

```
getNumeros(){  
  //retorna um Observable  
  const fonte = this.valService.getNumeros();  
  //adiciona 100 a cada número  
  const exemplo = fonte.pipe(  
    map(val => val + 100),  
  );  
  const inscricao = exemplo.subscribe(  
    numero => console.log(numero)  
  );  
  // 101, 102, 103, 104, 105  
}
```

Aula10\1.1_observable\src\app\home\home.component.ts

debounceTime

Neste código,
debounceTime(1000) irá atrasar
a emissão de valores da fonte
Observable em 1 segundo

```
getNumeros(){  
    //retorna um Observable  
    const fonte = this.valService.getNumeros();  
    //adiciona 100 a cada número  
    const exemplo = fonte.pipe(  
        debounceTime(1000),  
        map(val => val + 100),  
    );  
    const inscricao = exemplo.subscribe(  
        numero => console.log(numero)  
    );  
    // 101, 102, 103, 104, 105  
}
```

filter

```
getNumeros(){  
  //retorna um Observable  
  const fonte = this.valService.getNumeros();  
  //adiciona 100 a cada número  
  const exemplo = fonte.pipe(  
    map(val => val + 100),  
    filter(n => n % 2 === 0)  
  );  
  const inscricao = exemplo.subscribe(  
    numero => {  
      console.log(numero);  
      this.numeros = numero;  
    }  
  );  
  // 101, 102, 103, 104, 105  
}
```


mergeMap

```
getNumeros(){  
  //retorna um Observable  
  const fonte = this.valService.getNumeros();  
  //adiciona 100 a cada número  
  const exemplo = fonte.pipe(  
    map(val => val + 100),  
    filter(n => n % 2 === 0),  
    mergeMap(val => of(`${val} alterado!`)),  
  );  
  const inscricao = exemplo.subscribe(  
    numero => {  
      console.log(numero);  
    }  
  );  
  // 101, 102, 103, 104, 105  
}
```


Subject

- Os **subjects** são um tipo especial de **Observable** que atua tanto como **Observer** quanto como **Observable**. Eles permitem valores multicast para vários observadores.
 - Enquanto os Observables normais são unicast (cada Observer inscrito tem sua própria execução do Observable), os Subjects são multicast.
- Os **subjects** são úteis para cenários em que diversas partes de um aplicativo precisam se comunicar e reagir a eventos em tempo real.
- Eles são como um EventEmitter, que mantém uma lista de muitos ouvintes;

Observable vs Subject

Observable	Subject
Unicast: Cada assinatura obtém sua própria execução independente do Observable	Multicast: Quando um valor é emitido, ele é passado para todos os observadores ao mesmo tempo.
Não possui métodos como next(), error() e complete().	Possui métodos como next(), error() e complete(), permitindo controlar manualmente quando o Assunto emite um valor ou erro, ou quando é concluído.
Começa a emitir valores somente quando alguém se inscreve.	Pode começar a emitir valores imediatamente, e qualquer observador que se inscrever posteriormente receberá o último valor emitido pelo Subject
Observables são os blocos básicos de construção da programação reativa com RxJS.	Os Subjects são um tipo especial de Observável que é mais avançado e oferece mais funcionalidades.

Subjects

```
export class AutenticacaoService {  
  
    private autenticacaoUsuarioSubject = new Subject<boolean>();  
  
    //O $ após uma variável é uma convenção de nomenclatura  
    // comum em programação reativa e Angular  
    //quando se trabalha com Observables  
  
    usuarioAutenticado$ = this.autenticacaoUsuarioSubject.asObservable();  
  
    constructor() { }  
      
    autenticarUsuario(estaAutenticado: boolean) {  
        this.autenticacaoUsuarioSubject.next(estaAutenticado);  
    }  
}
```

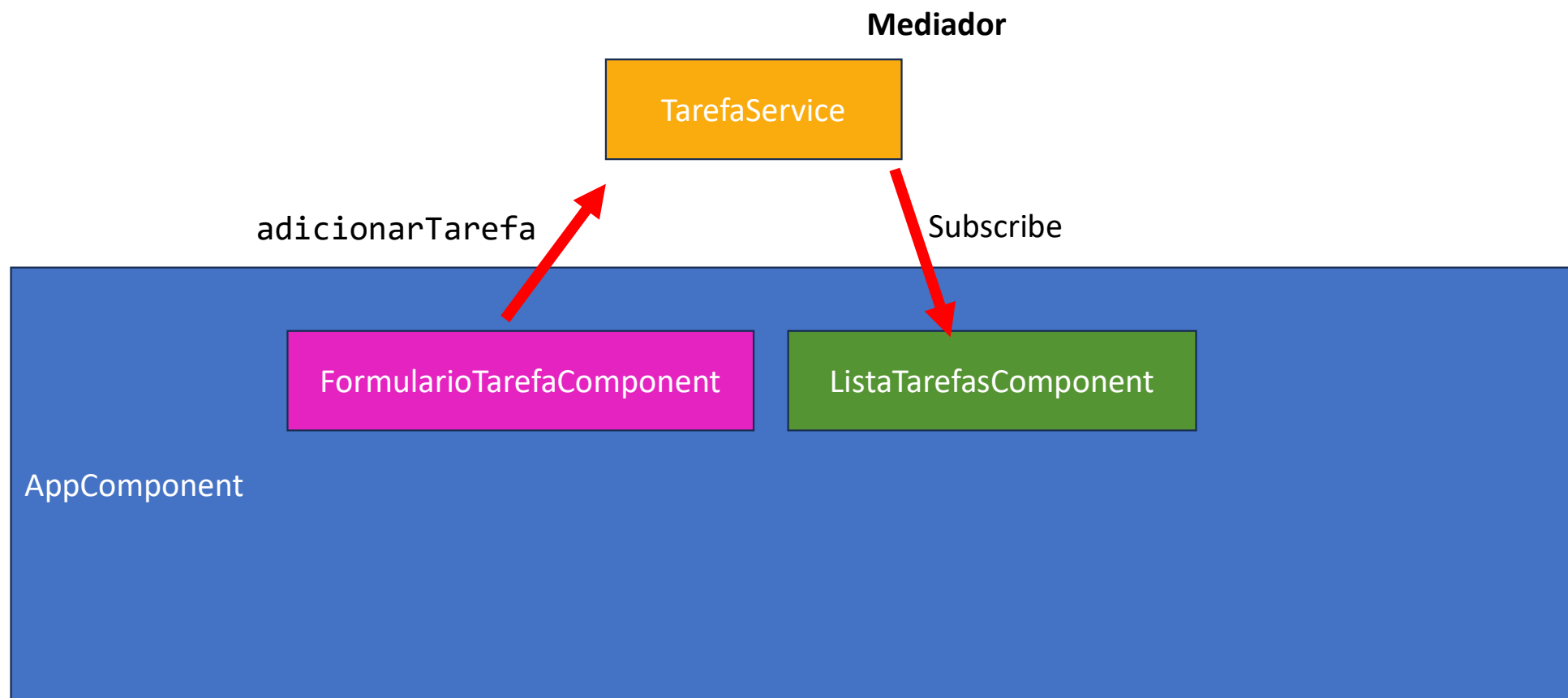
Subjects

```
constructor(private autenticaService: AutenticacaoService) {  
  this.autenticaService.usuarioAutenticado$.subscribe(estaAutenticado => {  
    console.log('usuarioAutenticado:', estaAutenticado);  
    this.autenticado = estaAutenticado;  
  });  
}  
  
autenticar() {  
  this.autenticaService.autenticarUsuario(true);  
}  
  
desautenticar() {  
  this.autenticaService.autenticarUsuario(false);  
}
```

Comunicação entre componentes não relacionados

- O Angular oferece várias técnicas para compartilhar dados entre componentes com algum nível de relacionamento;
 - Pai para filho @Input
 - Filho Pai @Output, EventEmitter
- Como fazer que componentes não relacionados se comuniquem de forma eficiente?
- Utilizando um Service que atua como um mediador para a comunicação entre componentes não relacionados
 - providedIn: 'root'

Aplicação Lista de Tarefas



Exercicio01

- Crie um serviço que retorne um observable que emita números de 1 a 100;
- Crie um componente que se inscreva nesse observable e imprima os números no seu view;

Referências

- <https://rxjs.dev/guide/operators>
- <https://www.linkedin.com/pulse/top-9-commonly-used-rxjs-operators-angular-akash-chauhan/>
- <https://www.learnrxjs.io/learn-rxjs/operators>
- <https://rxjs.dev/guide/observable>
- <https://rxjs.dev/guide/subject>