Pipes Personalizados de Conversão de Status no Angular 20

1. 📌 O que são Pipes?

- **Pipes** em Angular são **transformadores de dados** usados diretamente no template.

Exemplo clássico:

{{ today | date:'shortDate' }}

- Converte um objeto `Date` para um formato curto.

Quando precisamos **criar transformações específicas** (ex.: converter status de usuário), usamos **pipes personalizados**.

---

2. 📌 Criando um Pipe Básico de Conversão de Status

Exemplo: Converter status numérico em texto

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'status',
  standalone: true
})
export class StatusPipe implements PipeTransform {
  transform(value: number): string {
    switch (value) {
      case 0: return 'Inativo';
      case 1: return 'Ativo';
      case 2: return 'Pendente';
      default: return 'Desconhecido';
    }
  }
}
```

Uso no template:

```
<p>Status: {{ 1 | status }}</p>
<!-- Saída: Status: Ativo -->
```

---

3. 📌 Pipes com Argumentos

Podemos passar **parâmetros extras** para personalizar a conversão.

```
@Pipe({
  name: 'status',
  standalone: true
})
export class StatusPipe implements PipeTransform {
  transform(value: number, withEmoji: boolean = false): string {
    switch (value) {
      case 0: return withEmoji ? '❌ Inativo' : 'Inativo';
      case 1: return withEmoji ? '✅ Ativo' : 'Ativo';
      case 2: return withEmoji ? '⏳ Pendente' : 'Pendente';
      default: return '❓ Desconhecido';
    }
  }
}
```

Uso no template:

```
<p>Status: {{ 2 | status:true }}</p>
<!-- Saída: Status: ⏳ Pendente -->
```

---

4. 📌 Pipes Assíncronos (Avançado)

Um pipe pode lidar com **dados vindos de APIs** ou **Observables**.
Exemplo: converter um `idStatus` em texto **via chamada a serviço**.

```
import { Pipe, PipeTransform } from '@angular/core';
import { StatusService } from './status.service';
import { map, Observable } from 'rxjs';

@Pipe({
  name: 'statusAsync',
  standalone: true
})
export class StatusAsyncPipe implements PipeTransform {
  constructor(private statusService: StatusService) {}
```

```
  transform(value: number): Observable<string> {
    return this.statusService.getStatus(value).pipe(
      map(status => status?.label ?? 'Desconhecido')
    );
  }
}
```

Uso no template:

```
<p>Status: {{ user.statusId | statusAsync | async }}</p>
```

---

5. 📌 Pipes de Conversão Complexa

Um pipe pode formatar **cores, classes CSS ou ícones**.

```
@Pipe({
  name: 'statusClass',
  standalone: true
})
export class StatusClassPipe implements PipeTransform {
  transform(value: number): string {
    switch (value) {
      case 0: return 'bg-red-500 text-white';
      case 1: return 'bg-green-500 text-white';
      case 2: return 'bg-yellow-500 text-black';
      default: return 'bg-gray-400 text-black';
    }
  }
}
```

Uso no template:

```
<span [class]="user.status | statusClass">
  {{ user.status | status:true }}
</span>
```

---

6. 📌 Pipes Puros vs Impuros

- **Puro (default):** Só é recalculado quando a entrada muda.

- **Impuro:** Recalcula em cada ciclo de mudança → útil quando os dados mudam dentro de objetos/arrays.

```
@Pipe({
  name: 'status',
  pure: false, // recalcula sempre
  standalone: true
})
export class StatusPipe implements PipeTransform {
  transform(value: number): string {
    return value === 1 ? 'Ativo' : 'Inativo';
  }
}
```

---

7. 📌 Boas Práticas

✅ Use Pipes para **apresentação**, não para lógica pesada.
✅ Prefira Pipes **puros** para performance.
✅ Centralize conversões repetitivas em **pipes reutilizáveis**.
✅ Combine com **directives** e **components** para maior flexibilidade.

---

8. 📌 Casos Avançados de Uso

- **Internacionalização**: pipe que traduz status dependendo da linguagem (`pt`, `en`, etc).

- **Formatação dinâmica**: converter status para texto + cor + ícone.

Encadeamento de pipes:

```
 {{ 1 | status:true | uppercase }}
<!-- Saída: ✅ ATIVO -->
```

-
- **Uso em formulários**: exibir labels legíveis em `select` ou `mat-option`.

---

9. 📌 Exemplo Completo – Status com Ícone + Estilo

```
@Pipe({
  name: 'statusLabel',
  standalone: true
})
export class StatusLabelPipe implements PipeTransform {
  transform(value: number): { label: string; icon: string; color: string } {
    switch (value) {
      case 0: return { label: 'Inativo', icon: '❌', color: 'red' };
      case 1: return { label: 'Ativo', icon: '✅', color: 'green' };
      case 2: return { label: 'Pendente', icon: '⏳', color: 'orange' };
      default: return { label: 'Desconhecido', icon: '❓', color: 'gray' };
    }
  }
}
```

Uso no template:

```
<div *ngIf="user.status | statusLabel as status">
  <span [style.color]="status.color">{{ status.icon }} {{ status.label }}</span>
</div>
```

✅ Conclusão

Com Pipes personalizados no Angular 20 é possível:

- Converter códigos em textos legíveis.
- Criar representações visuais (ícones, cores, classes CSS).
- Integrar com **APIs e Observables**.
- Encadear pipes para transformações complexas.
- Tornar a UI **mais declarativa, legível e reutilizável**.
