PIPES & DYNAMIC COMPONENTS

PIPES:

Transforming the values is what we use pipes for.

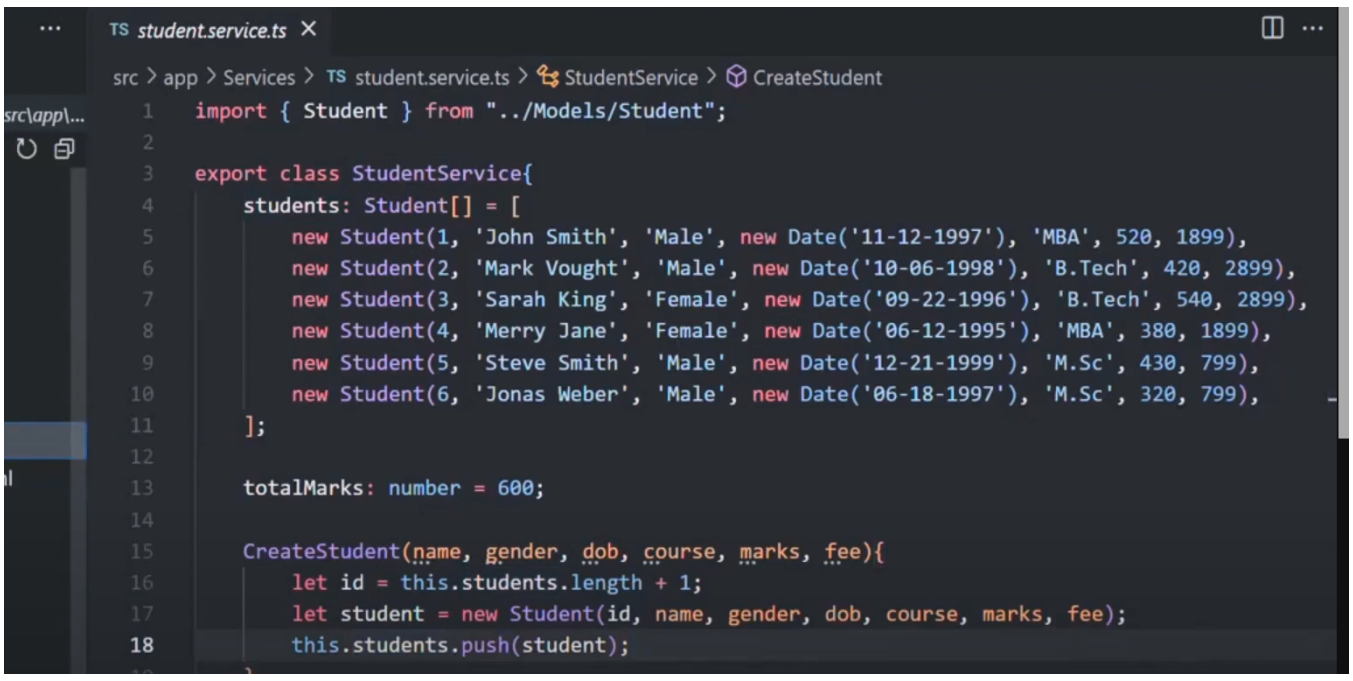
so, pipes in angular are used to transform data before displaying it in the view in simple terms we can say that angular pipes takes data as an input and formats or transform the data before displaying it in the template.

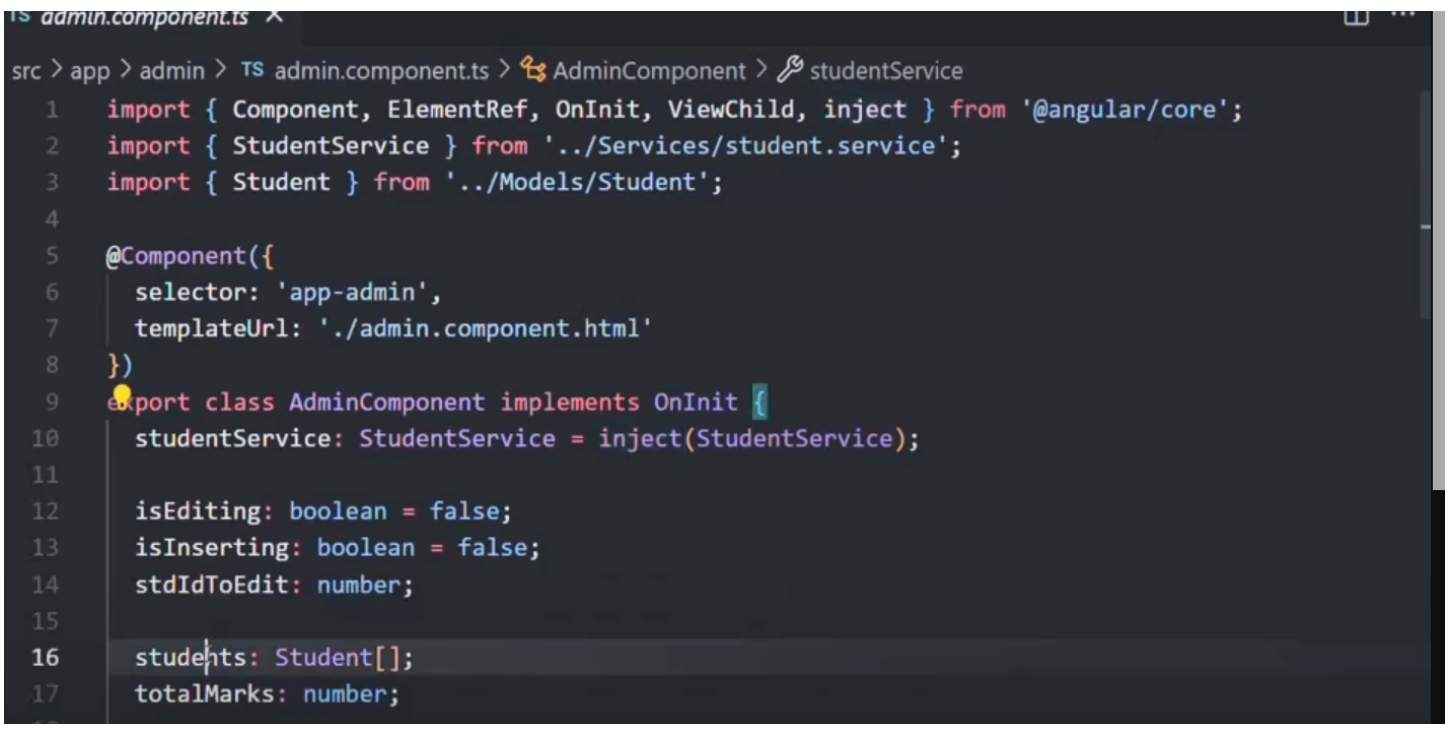
Here's the list of API pipes presented side by side for easier reading:

- AsyncPipe | CurrencyPipe | DATE_PIPE_DEFAULT_OPTIONS
- DATE_PIPE_DEFAULT_TIMEZONE | DatePipe | DatePipeConfig
- DecimalPipe | I18nPluralPipe | I18nSelectPipe
- JsonPipe | KeyValuePipe | LowerCasePipe
- PercentPipe | SlicePipe | TitleCasePipe
- UpperCasePipe

What is a pipe?

Pipes in Angular are used to transform or format data before displaying it in the view. Angular pipe takes data as an input and it formats or transforms that data before displaying it in the template.





Pipes in Angular

Overview of Pipes

- **Definition**: Pipes are simple functions that accept an input value and return a transformed value. They are used in templates to format data displayed to the user.
- **Purpose**: Enhance the presentation of data without changing the underlying data model.

Types of Pipes

1. **Built-in Pipes**
    - Angular provides several built-in pipes that cover common transformation needs.
    - **Common Built-in Pipes**:

- **CurrencyPipe**: Formats a number as currency.

```html
{{ amount | currency:'USD':'symbol' }}
```

- **DatePipe**: Formats a date value according to locale rules.

```html
{{ today | date:'fullDate' }}
```

- **DecimalPipe**: Formats a number as decimal.

```html
{{ value | number:'1.2-2' }}
```

- **JsonPipe**: Converts an object to a JSON string.

```html
{{ object | json }}
```

- **SlicePipe**: Creates a new array or string containing a subset of the elements.

```html
{{ array | slice:1:3 }}
```

- **UpperCasePipe**: Transforms text to uppercase.

```html
{{ text | uppercase }}
```

- **LowerCasePipe**: Transforms text to lowercase.

```html
{{ text | lowercase }}
```

2. **Custom Pipes**

   - You can create your own pipes to handle specific transformation logic.
   - **Creating a Custom Pipe**:

     1. Generate a pipe using Angular CLI:

```bash
ng generate pipe myCustomPipe
```

2. Implement the transformation logic.

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

@Pipe({
  name: 'myCustomPipe'
})
export class MyCustomPipe implements PipeTransform {
  transform(value: string): string {
    return value.toUpperCase(); // Example transformation
  }
}
```

3. Use the custom pipe in a template:

```html
{{ 'hello' | myCustomPipe }}
```

Pipe Parameters

- Pipes can accept additional parameters to customize the transformation.
  - **Example with Parameters**:
    - Custom pipe that formats a string to a specified length:

```
@Pipe({
  name: 'truncate'
})
export class TruncatePipe implements PipeTransform {
  transform(value: string, limit: number): string {
    return value.length > limit ? value.substring(0, limit) + '...' : value;
  }
}
```

- Usage in template:

```html
{{ longText | truncate:10 }}
```

Async Pipe

- **Definition**: Automatically subscribes to an Observable or Promise and returns the latest value.
- **Usage**:
  - Example in a template:

```html
<div *ngIf="data$ | async as data">
  {{ data }}
</div>
```

Key Takeaways

- Pipes are a powerful feature in Angular for transforming data in templates.
- Built-in pipes cover many common scenarios, while custom pipes can handle specific needs.
- Pipes can accept parameters to fine-tune their behavior.

Next Steps

- Experiment with built-in and custom pipes in Angular applications to deepen understanding and practice their usage.

How to Create Custom Pipes?

How to create a custom pipe?

We can create a custom angular pipe in three simple steps:

1. Create a typescript class and export it. By convention, a pipe class name should end with Pipe.
2. Decorate that class with `@Pipe` Decorator. There we can specify a name for the pipe.
3. Inherit `PipeTransform` interface and implement its `transform` method.

admin.component.html | percentage.pipe.ts | app.module.ts

src > app > Pipes > percentage.pipe.ts > PercentagePipe

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


@Pipe({
    name: 'percentage'
})
export class PercentagePipe implements PipeTransform{
    transform(value: any) {
        return value * 100;
    }
}
```

admin.component.html 1 | percentage.pipe.ts | app.module.ts

src > app > Pipes > percentage.pipe.ts > PercentagePipe > transform

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


@Pipe({
    name: 'percentage'
})
export class PercentagePipe implements PipeTransform{
    transform(value: number, total: number) {
        return value / total * 100;
    }
}
```

```ts
@NgModule({
  declarations: [
    AppComponent,
    AdminComponent,
    PercentagePipe
  ]
```

Pure & Impure Pipes in Angular

What is a pure pipe?

A pure pipe is that pipe which gets called whenever there is a pure change on the input value. By default every pipe is a pure pipe.

```
@Pipe({
    name: 'filter',
    pure: true
})
```

What is pure change?

Following changes are considered as pure change in a data:

1. If the pipe is being used on a primitive type like string, number, boolean etc. And if the value of that primitive type changes, it is considered as a pure change
2. If the pipe is being used on a reference type input, and if the reference of that input changes, than the change is pure change.
3. **NOTE**: But if the input is of reference type, and only its property value changes and its reference has not changed, that change is not a pure change.

What is an impure pipe?

A pipe is impure pipe if it is not pure. An impure pipe gets called for each change detection cycle and it is performance intensive.

```
@Pipe({
    name: 'filter',
    pure: false
})
```

Pure and Impure Pipes in Angular

Overview

- **Pipes** in Angular can be classified into two categories: **pure pipes** and **impure pipes**. This classification affects how they handle data changes and re-evaluation.

Pure Pipes

- **Definition**: Pure pipes are those that only re-evaluate when their input values change. Angular performs change detection for pure pipes only when their input references change.

- **Characteristics**:
  - Efficient and performant.
  - Ideal for scenarios where the output is predictable based on the input.
- **Example of a Pure Pipe**:

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

@Pipe({
  name: 'purePipe',
  pure: true // This is the default setting
})
export class PurePipe implements PipeTransform {
  transform(value: number): number {
    return value * 2; // Example transformation
  }
}
```

- **Usage**:

```html
<p>{{ 5 | purePipe }}</p> <!-- Output: 10 -->
```

Impure Pipes

- **Definition**: Impure pipes are those that re-evaluate on every change detection cycle, regardless of whether their input values have changed.
- **Characteristics**:
  - Can lead to performance issues if not used carefully.
  - Useful when you need to respond to changes in data that are not directly tied to the input parameters (e.g., a changing state).
- **Example of an Impure Pipe**:

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

@Pipe({
  name: 'impurePipe',
  pure: false // Explicitly marked as impure
})
export class ImpurePipe implements PipeTransform {
  transform(value: number): number {
    console.log('Impure pipe evaluated'); // Logs every change detection cycle
    return value * 2;
  }
}
```

- **Usage**:

```html
<p>{{ counter | impurePipe }}</p> <!-- Logs on every change detection -->
```

When to Use Each

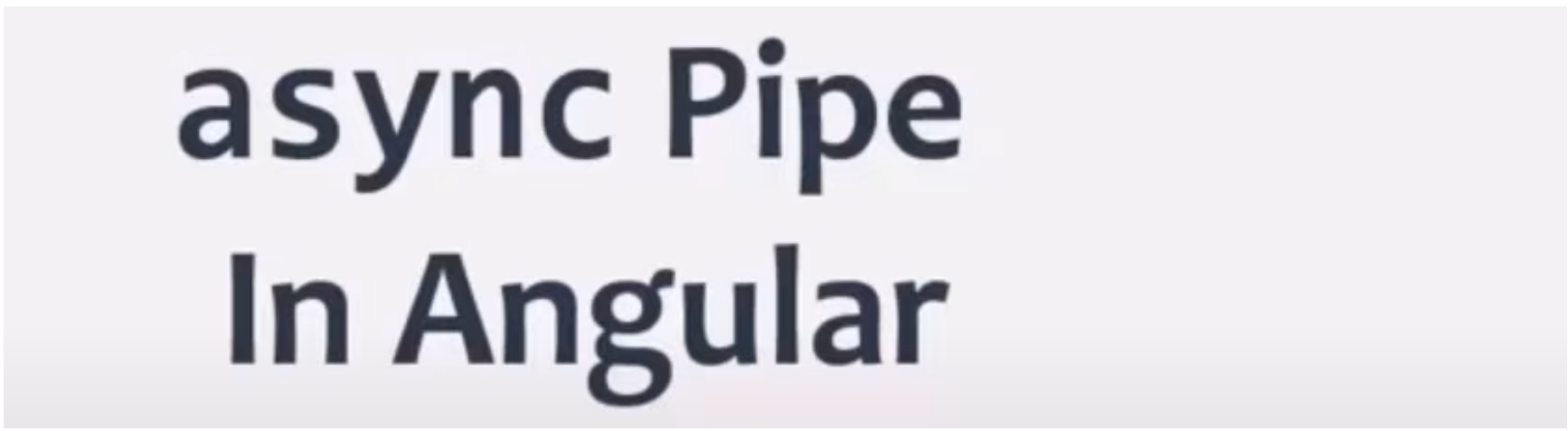
- **Use Pure Pipes**:
  - When the output is solely dependent on the input.
  - For performance-sensitive applications to minimize unnecessary computations.
- **Use Impure Pipes**:
  - When you need to react to changes that are not tied to the input values.
  - When your pipe needs to observe changes outside of its input parameters.

Example Scenario

- **Pure Pipe Example**:
  - A pure pipe to format a number as currency, where the formatting is solely dependent on the number input.
- **Impure Pipe Example**:
  - An impure pipe that fetches the current time or generates a random number on every change detection, reacting to changes in the application state.

Conclusion

- Understanding the difference between pure and impure pipes is essential for optimizing Angular applications. Using pure pipes where appropriate can lead to significant performance improvements. Impure pipes should be used judiciously to avoid unnecessary evaluations.



async pipe

The async pipe allows us to handle asynchronous data. It allows us to subscribe to an observable or promise from the view template and returns the value emitted.

```
totalStudents = new Promise((resolve, rejection) => {
  setTimeout(() => {
    resolve(this.students.length);
  }, 2000);
});
```

Async Pipe in Angular

Overview

- The **async pipe** is a built-in Angular pipe that allows you to subscribe to an `Observable` or `Promise` and automatically manage the subscription for you.
- It simplifies handling asynchronous data in templates, eliminating the need to manually subscribe and unsubscribe from Observables.

Key Features

- Automatically subscribes to an `Observable` or `Promise`.
- Unsubscribes automatically when the component is destroyed, preventing memory leaks.
- Updates the template with the latest emitted value from the Observable.

Usage

1. **In the Template**: Use the `async` pipe in the template expression.
2. **Data Source**: It can be used with `Observable` from Angular's `RxJS` library or with `Promise`.

Example with Observable

Step 1: Create a Service

```
import { Injectable } from '@angular/core';
import { Observable, of } from 'rxjs';
import { delay } from 'rxjs/operators';

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

Copy code

```
export class DataService {
  getData(): Observable<string> {
    return of('Hello, Async Pipe!').pipe(delay(2000)); // Simulates async data
  }
}
```

Step 2: Use in a Component

typescript Copy code

```typescript
import { Component, OnInit } from '@angular/core';
import { DataService } from './data.service';

@Component({
  selector: 'app-example',
  template: `<p>{{ data$ | async }}</p>` // Async pipe in template
})
export class ExampleComponent implements OnInit {
  data$: Observable<string>;

  constructor(private dataService: DataService) {}

  ngOnInit() {
    this.data$ = this.dataService.getData(); // Assign Observable to data$
  }
}
```

Example with Promise

Step 1: Modify the Service

typescript Copy code

```typescript
getPromiseData(): Promise<string> {
  return new Promise(resolve => {
    setTimeout(() => resolve('Hello, Async Pipe with Promise!'), 2000);
  });
}
```

Step 2: Use in the Component

typescript Copy code

```typescript
export class ExampleComponent implements OnInit {
  promiseData: Promise<string>;

  ngOnInit() {
    this.promiseData = this.dataService.getPromiseData(); // Assign Promise to promiseData
  }
}
```

Template Usage:

```html
<p>{{ promiseData | async }}</p> <!-- Async pipe with Promise -->
```

Benefits

- Simplifies template syntax for asynchronous data.
- Reduces boilerplate code for managing subscriptions.
- Automatically handles cleanup on component destruction.

Considerations

- The async pipe can only be used with Observables and Promises.
- Ensure the data source returns the expected type to avoid template errors.

Conclusion

The async pipe is a powerful feature in Angular that enhances the management of asynchronous data in templates, leading to cleaner and more maintainable code. Use it to simplify your handling of Observables and Promises without the hassle of manual subscription management.