



# ANGULAR 6

Lesson 05



The background of the slide features a dark blue field with large, organic, red shapes that overlap and curve across the space. A red L-shaped graphic is positioned on the right side, consisting of a vertical bar and a horizontal bar meeting at a corner.

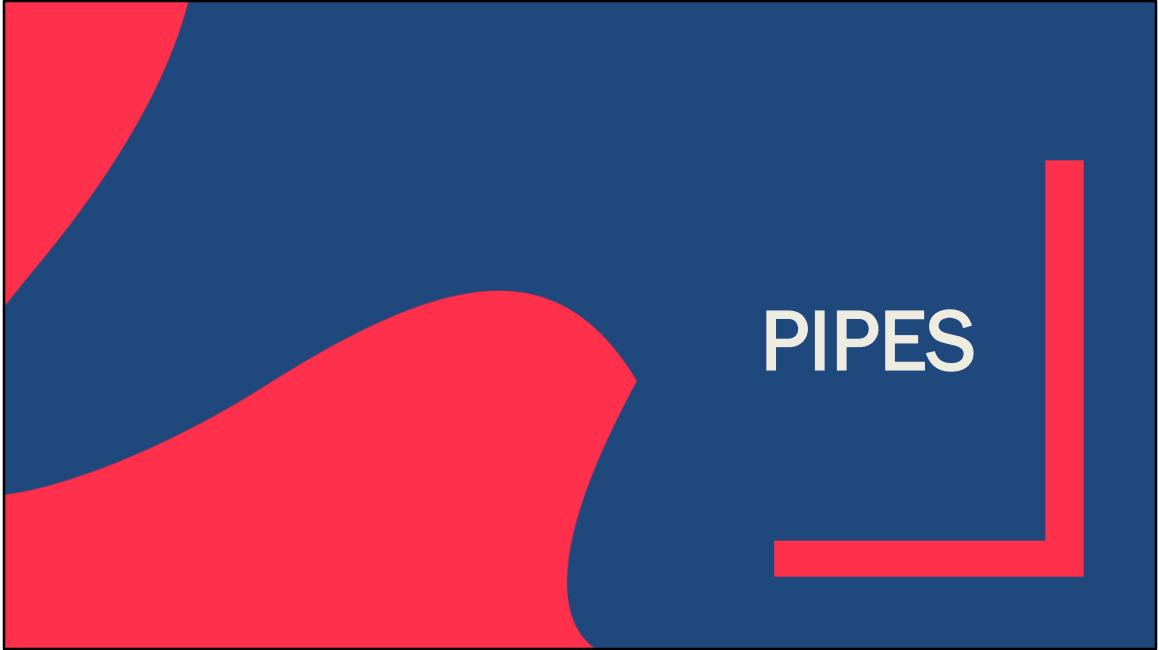
# PIPES, SERVICES & DEPENDENCY INJECTION

Objectives

# Pipes, Services & Dependency Injection

- Parametrized Pipes
- Chaining Multiple Pipes
- Creating a Custom Pipe
- Creating a Filter Pipe
- Pure and Impure Pipes (or: How to "fix" the Filter Pipe)
- Understanding the "async" Pipe
- Services
- Dependency Injections
- Creating Data Service
- Understanding Hierarchical Injector
- Services for Cross Component Communication
- Injection Tokens





Add instructor notes here.

## Pipe

A pipe takes in data as input and transforms it to a desired output.

Pipes transform bound properties before they are displayed

Angular pipes, a way to write display-value transformations that we can declare in your HTML.

To pass an argument to a pipe in the HTML form, pass it with a colon after the pipe (for multiple arguments, simply append a colon after each argument)

Angular gives us several built-in pipe like lowercase, date, number, decimal, percent, currency, json, slice etc

Angular provides a way to create custom pipes as well.



















Pipes are used when the data is not in a format appropriate for display

# Build in Pipes

<https://angular.io/api?type=pipe> We Can see all Build in Pipes

## common

 AsyncPipe	 CurrencyPipe	 DatePipe
 DecimalPipe	 DeprecatedCurrencyPipe	 DeprecatedDatePipe
 DeprecatedDecimalPipe	 DeprecatedPercentPipe	 I18nPluralPipe
 I18nSelectPipe	 JsonPipe	 LowerCasePipe
 PercentPipe	 SlicePipe	 TitleCasePipe
 UpperCasePipe		



Angular comes with a stock of pipes such as DatePipe, UpperCasePipe, LowerCasePipe, CurrencyPipe, and PercentPipe. They are all available for use in any template. Angular doesn't have a FilterPipe or an OrderByPipe

## CURRENCYPIPE

This pipe is used for formatting currencies. Its first argument is an abbreviation of the currency type (e.g. "EUR", "USD", and so on), like so:

```
{{ 1234.56 | currency:'CAD' }}
```

The above prints out CA\$1,234.56.

## Chaining Pipes & Parameterizing a pipe

- We can chain pipes together in potentially useful combinations.
- A pipe may accept any number of optional parameters to fine-tune its output.
- We add parameters to a pipe by following the pipe name with a colon ( : ) and then the parameter value (e.g., currency:'EUR').
- If our pipe accepts multiple parameters, we separate the values with colons (e.g. slice:1:5).

```
<tr *ngFor="let emp of employess" >
<td>{{emp.empld}}</td>
<td>{{emp.empName | uppercase | slice:1:3}}</td> <!-- Channing pipes & passing parameter -->
<td>{{emp.empSal | currency:'USD':true}}</td> <!-- Currency pipes -->
<td>{{emp.empDep}}</td>
<td>{{emp.empjoiningdate | date:'fullDate'|uppercase}}</td> <!-- Dates pipes channing pipes -->
```



A pipe can accept any number of optional parameters to fine-tune its output. To add parameters to a pipe, follow the pipe name with a colon ( : ) and then the parameter value (such as currency:'EUR'). If the pipe accepts multiple parameters, separate the values with colons (such as slice:1:5)

You can chain pipes together in potentially useful combinations. In the following example, to display the birthday in uppercase, the birthday is chained to the DatePipe and on to the UpperCasePipe. The birthday displays as **APR 15, 1988**. The chained hero's birthday is {{ birthday | date | uppercase }}

## Creating a Custom Pipe

- A pipe is a class decorated with pipe metadata.
- The pipe class implements the PipeTransform interface's transform method that accepts an input value followed by optional parameters and returns the transformed value.
- There will be one additional argument to the transform method for each parameter passed to the pipe. Your pipe has one such parameter: the exponent.





## Creating a Custom Pipe (Contd...)



- To tell Angular that this is a pipe, you apply the `@Pipe` decorator, which you import from the core Angular library.
- The `@Pipe` decorator allows you to define the pipe name that you'll use within template expressions. It must be a valid JavaScript identifier.

```
import {PipeTransform,Pipe} from 'angular2/core';

@Pipe({ name : 'customPipe'})

export class ExponentialStengthPipe implements PipeTransform{
  transform(value:number,args:string[]):any {
    return Math.pow(value,parseInt(args[0] || '1', 10));
  }
}
```



In share/pipes folder, create a new .ts file named **trim.pipe.ts**

Import from 'angular/core' the module **Pipe** and **PipeTransform**. We tell Angular that this is a pipe by applying the `@Pipe` decorator which we import from the core Angular library.1

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

Creating The pipe class implements the **PipeTransform** interface's transform method that accepts an input value followed by optional parameters and returns the transformed value.

The **@Pipe** decorator allows us to define the pipe name that we'll use within template expressions. It must be a valid JavaScript identifier. We should be always use the **"PipeTransform"** interface, which forces us to have a **transform()** method.

*The **transform** method is essential to a pipe. The **PipeTransform** interface defines that method and guides both tooling and the compiler. It is technically optional; Angular looks for and executes the transform method regardless.*

### How to use a Custom Pipe

1. We use our custom pipe the same way we use the built-in pipes.
2. We must include our pipe in the pipes array of the `@Component` decorator.

Add instructor notes  
here.

# Demo

- Demo Build In Pipes
- Demo Custom Pipe

Cap



Add the notes here.



# SERVICES & DEPENDENCY INJECTION

# Dependency Injection

## ■ Problem without DI

```
class Engine{  
    constructor(newparameter){}  
}  
class Tires{  
    constructor(){}
```

```
class Car{  
    engine;  
    tires;  
    constructor()  
    {  
        this.engine = new Engine();  
        this.tires = new Tires();  
    }  
}
```



The Car class creates everything it needs inside its constructor. What's the problem? The problem is that the Car class is brittle, inflexible, and hard to test.

This Car needs an engine and tires. Instead of asking for them, the Car constructor instantiates its own copies from the very specific classes Engine and Tires.

What if the Engine class evolves and its constructor requires a parameter? That would break the Car class and it would stay broken until you rewrote it along the lines of `this.engine = new Engine(theNewParameter)`. The Engine constructor parameters weren't even a consideration when you first wrote Car. You may not anticipate them even now. But you'll *have* to start caring because when the definition of Engine changes, the Car class must change. That makes Car brittle.

What if you want to put a different brand of tires on your Car? Too bad. You're locked into whatever brand the Tires class creates. That makes the Car class inflexible.

Right now each new car gets its own engine. It can't share an engine with other cars. While that makes sense for an automobile engine, surely you can think of other dependencies that should be shared, such as the onboard wireless connection to the manufacturer's service center. This Car lacks the flexibility to share services that have been created previously for other consumers.

When you write tests for Car you're at the mercy of its hidden dependencies. Is it even possible to create a new Engine in a test environment? What does Engine depend upon?

What does that dependency depend on? Will a new instance of Engine make an asynchronous call to the server? You certainly don't want that going on during tests.

What if the Car should flash a warning signal when tire pressure is low? How do you confirm that it actually does flash a warning if you can't swap in low-pressure tires during the test?

You have no control over the car's hidden dependencies. When you can't control the dependencies, a class becomes difficult to test.

How can you make Car more robust, flexible, and testable?

## Dependency Injection (Contd...)

- Dependency injection is an important application design pattern
- Angular has its own dependency injection framework
- DI allows to inject dependencies in different components across applications, without needing to know, how those dependencies are created, or what dependencies they need themselves.
- DI can also be considered as framework which helps us out in maintaining assembling dependencies for bigger applications.

```
class Car{
  engine;
  tires;
  constructor()
  {
    this.engine = new Engine();
    this.tires = new Tires();
  }
}
```

```
class Car{
  engine;
  tires;
  constructor(engine, tires)
  {
    this.engine = engine;
    this.tires = tires;
  }
}
```



The definition of the engine and tire dependencies are decoupled from the Car class. We can pass in any kind of engine or tires you like, as long as they conform to the general API requirements of an engine or tires. The Car class is much easier to test now because you are in complete control of its dependencies. It's a coding pattern in which a class receives its dependencies from external sources rather than creating them itself.

Anyone who wants a Car must now create all three parts: the Car, Engine, and Tires. The Car class shed its problems at the consumer's expense. You need something that takes care of assembling these parts.

## Dependency Injection (Contd...)

**Example 1 : DI**

```
public description = 'DI';  
constructor(public engine: Engine, public tires: Tires) { }
```

**Example 2 : DI**

```
class Engine2 {  
  constructor(public cylinders: number) { }  
}  
  
let bigCylinders = 12;  
let car = new Car(new Engine2(bigCylinders), new Tires());
```



Angular ships with its own dependency injection framework. This framework can also be used as a standalone module by other applications and frameworks.

Add instructor notes here.

## Services

- Services provided architectural way to encapsulate business logic in a reusable fashion
- Services allow to keep logic out of your components, directives and pipe classes
- Services can be injected in the application using Angular's dependency injection (DI)
- Angular has In-built service classes like Http, FormBuilder and Router which can be used to perform specific things that are non component specific.
- Custom Services are most often used to create Data Services.



A service is a class with a focused purpose. We often create a service to implement functionality that is independent from any particular component.

Services are used to share the data and logic across components or to encapsulate external interactions such as data access.

To create a service class there is no need to do anything angular specific, no Meta data, naming convention requirement or some functional interface that needs to be implemented. They're just plain old classes that you create to modularize reusable code.

Services are not only beneficial for modularity and single responsibility, but it also makes the code more testable.

Data service is a class that will handle getting and setting data from your data store

Add instructor notes here.

## Working with Services in Angular 6

- Component can work with service class using two ways
  - *Creating an instance of the service class*
    - Instances are local to the component, so data or other resources cannot be shared
    - Difficult to test the service
  - *Registering the service with angular using angular Injector*
    - Angular injector maintains a container of created service instances
    - The injector creates and manages the single instance or singleton of each registered service
    - Angular injector provides or injects the service class instance when the component is instantiated. This process is called dependency injection
    - Angular manages the single instance any data or logic in that instance is shared by that use it.
    - This technique is the recommended way to use services because it provides better service instances it allow sharing of data and other resources and it's easier to mock for testing purposes



So let's import Http, and we will also import the response class which we will need for some type checking. And both of these come from @angular/http. Now, we should also import the Http module into our app modules file. So, let's go ahead and do that before we forget. In our native modules at the top, I will import the HttpClientModule, and then down in our imports, let's include the HttpClientModule.

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
import { AppComponent } from './app.component';
import { EmployeeList } from './app.employee-list';
import { HomeComponent } from './HomeComponent';
import { EmployeeSearchComponent } from './EmployeeSearchComponent';
import { HttpClientModule } from '@angular/http';
import { Routes, RouterModule } from '@angular/router';
```

```
const appRoutes: Routes=[
    { path: '', redirectTo: '/getdata', pathMatch: 'full'},
    { path: 'getdata', component: EmployeeList },
    { path: 'postdata', component: HomeComponent },
    { path: 'search', component: EmployeeSearchComponent }
];
@NgModule({
  imports: [
    BrowserModule, FormsModule, HttpClientModule, RouterModule.forRoot(appRoutes) ],
  declarations: [
    AppComponent, EmployeeList, HomeComponent, EmployeeSearchComponent,
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

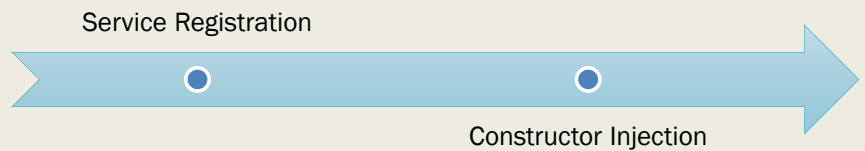


Add instructor notes here.

Cap

# Working with Services in Angular 6

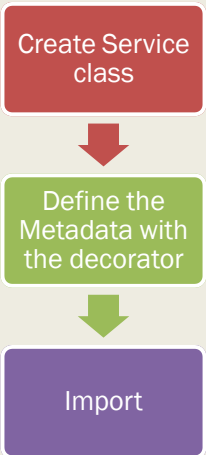
- Angular has dependency injection support baked into the framework which component directives in modular fashion.
- DI creates instances of objects and inject them into places where they are needed in the process.



Add instructor notes here.

## Building a Service

- Steps to build a service is similar to build components and a custom pipe.
- It is recommended that every service class use the injectable decorator for clarity and consistency.
- **@Injectable** is a decorator, that informs Angular 2 that the service has some dependencies itself. Basically services in Angular 2 are simple classes with the decorator @Injectable on top of the class, that provides a method to return some items.



In Angular, a service is basically any set of functionality that we want to be available to multiple components. It's just an easy way to wrap up some functionality. So inside of our app directory, let's create a projects service. And we'll call this projects.service.ts.

Now of course a service is not a component, so there's no need to import the component decorator. But there is another decorator that we need, and that is Injectable. So let's import Injectable from angular/core. Now as I said, Injectable is a decorator, and it doesn't take any properties. So we'll just call Injectable, and then export our class. We'll call the class ProjectsService.

# Steps to Create services

Create the Service File

Import the Injectable Member

- `import { Injectable } from '@angular/core';`

Add the Injectable Decorator

- `@Injectable()`

Export the Services Class

Import the Services to component

Add it as a Provider

- `providers: [ExampleService]`

Include it through dependency injection

- `constructor(private _exampleService: ExampleService) {}`

