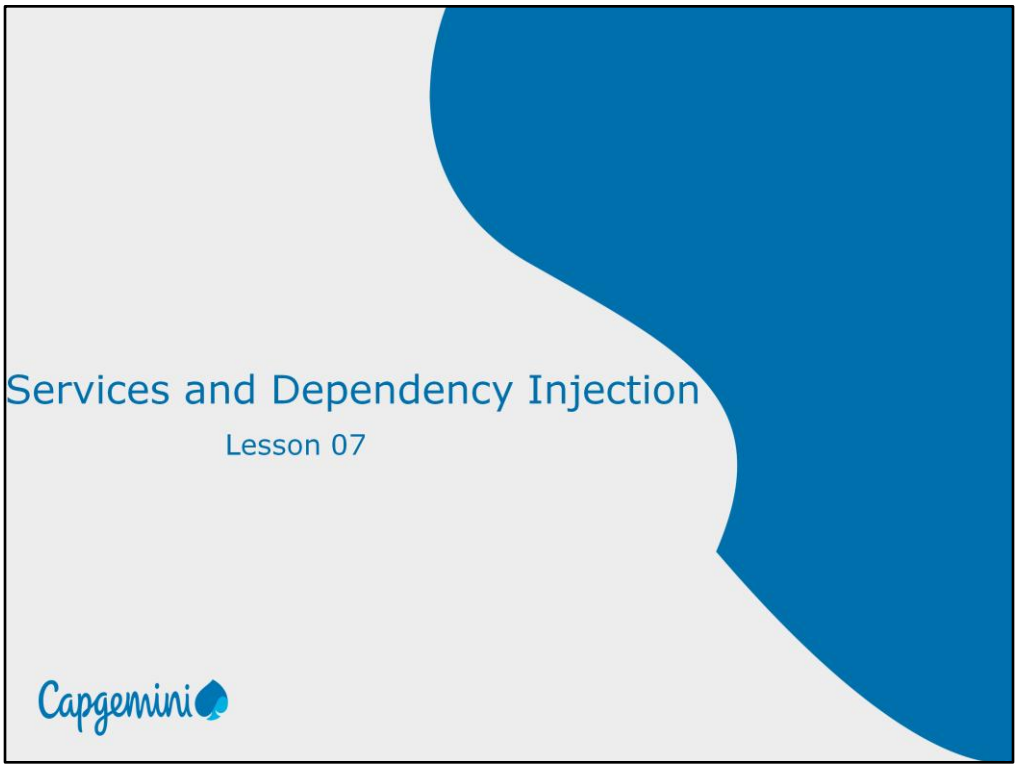
Instructor Notes:

Add instructor notes here.



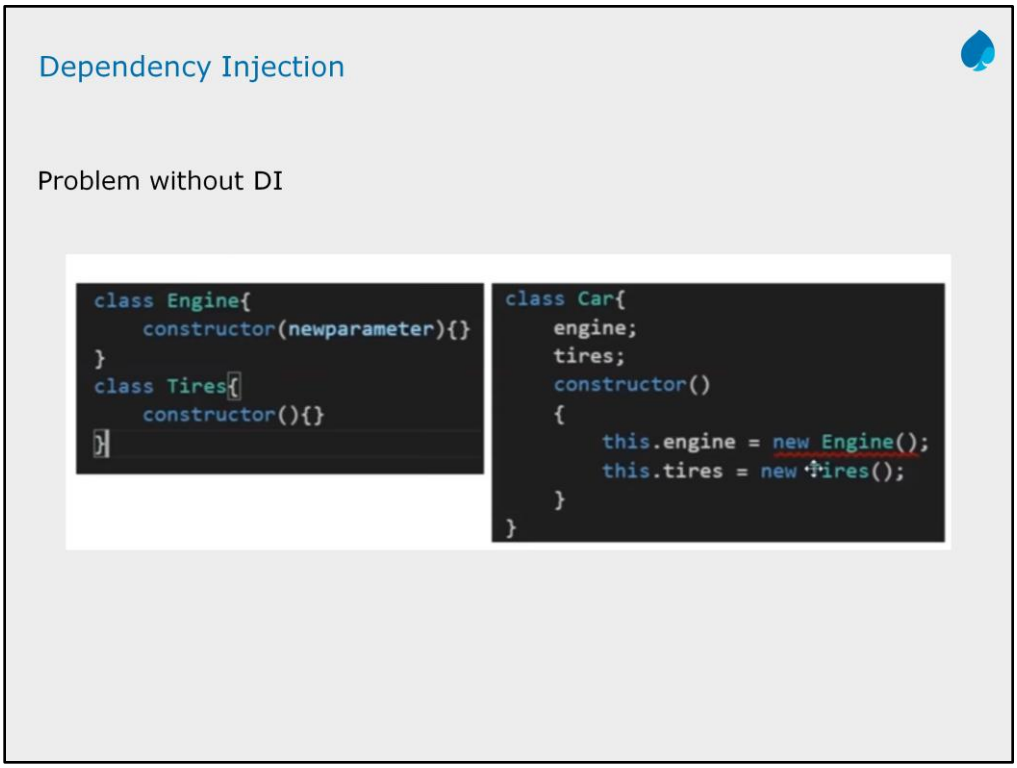
Instructor Notes:

Add instructor notes here.

Lesson Objectives

- What is a service?
- Injecting a service to a component
- Application wide dependency injection
- @Injectable classes
- Multiple service instances
- @Optional and @Host

Instructor Notes:



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 Carclass 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?

Instructor Notes:

Dependency Injection

- 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.

Without DI

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

With DI

```
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.

Instructor Notes:

Dependency Injection

```
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.

Instructor Notes:

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 contains logic for doing 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.

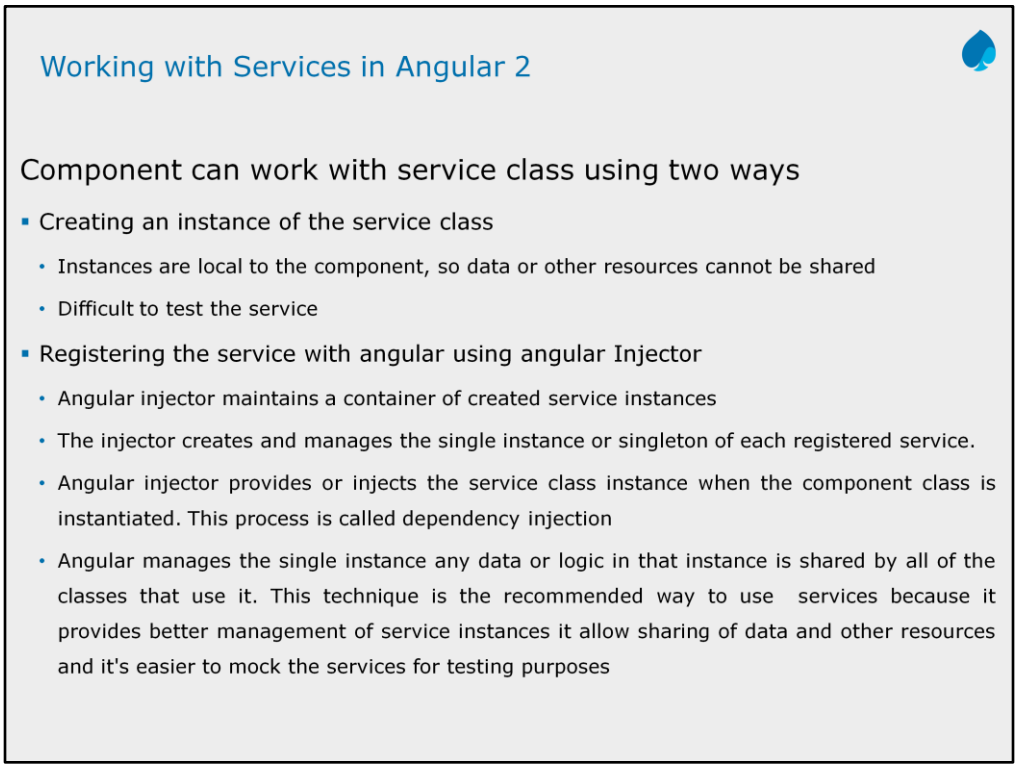
To create a service class there is no need to do anything angular specific, no Meta data, naming convention requirement or some functional interface that needs to be implemented. They're just plain old classes that you create to modularize reusable code.

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

Instructor Notes:

Add instructor notes here.



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 HttpModule, and then down in our imports, let's include the HttpModule.

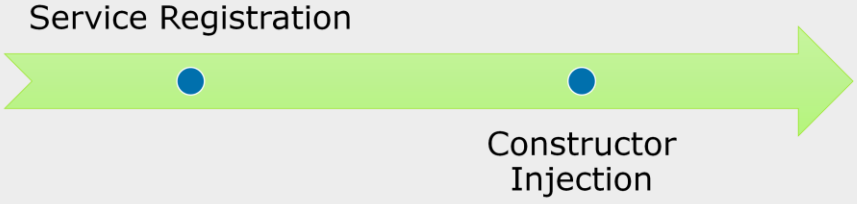
```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
import { AppComponent }  from './app.component';
import { EmployeeList }  from './app.employeelist';
import { HomeComponent }  from './HomeComponent';
import { EmployeeSearchComponent }  from './EmployeeSearchComponent';
import {HttpModule} 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,HttpModule,RouterModule.forRoot(appRoutes) ],
  declarations: [
AppComponent,EmployeeList,HomeComponent,EmployeeSearchComponent],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

Instructor Notes:

Add instructor notes here.

Working with Services in Angular 2

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



Instructor Notes:

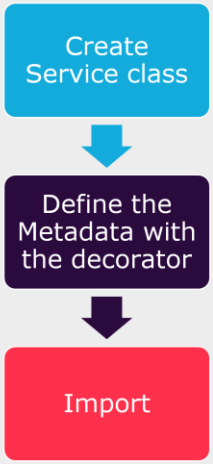
Add instructor notes here.

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.

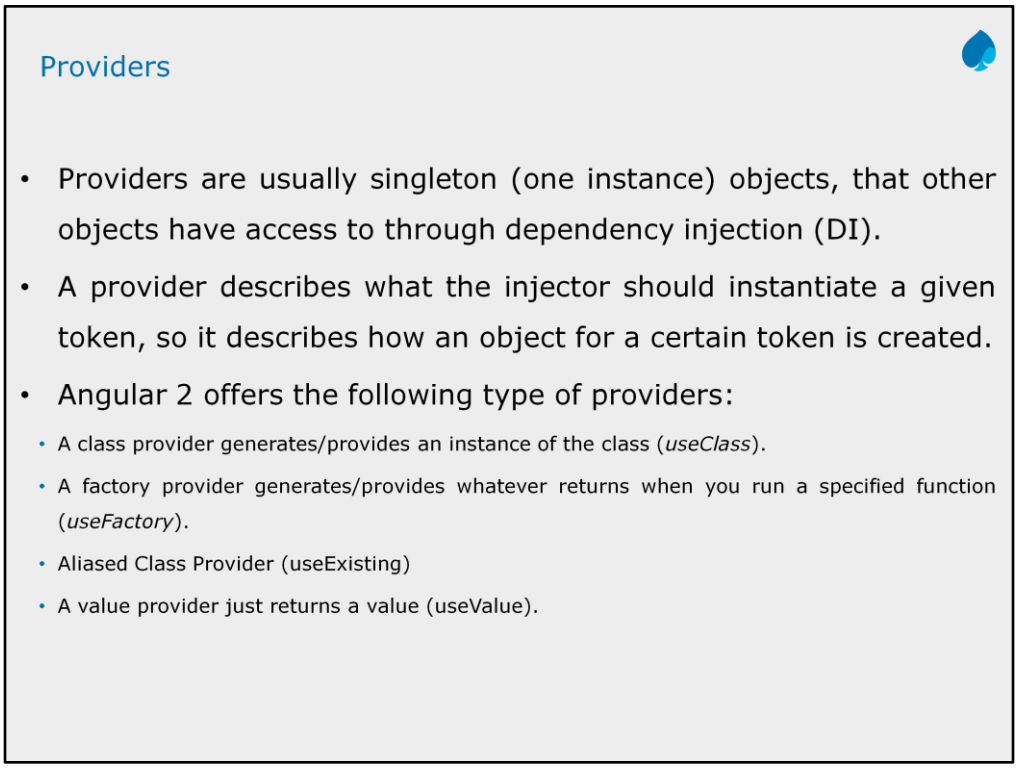
@Injectable is a decorator, that informs Angular 2 that the service has some dependencies itself. Basically services in Angular 2 are simple classes with the decorator @Injectable on top of the class, that provides a method to return some items.



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.

Instructor Notes:

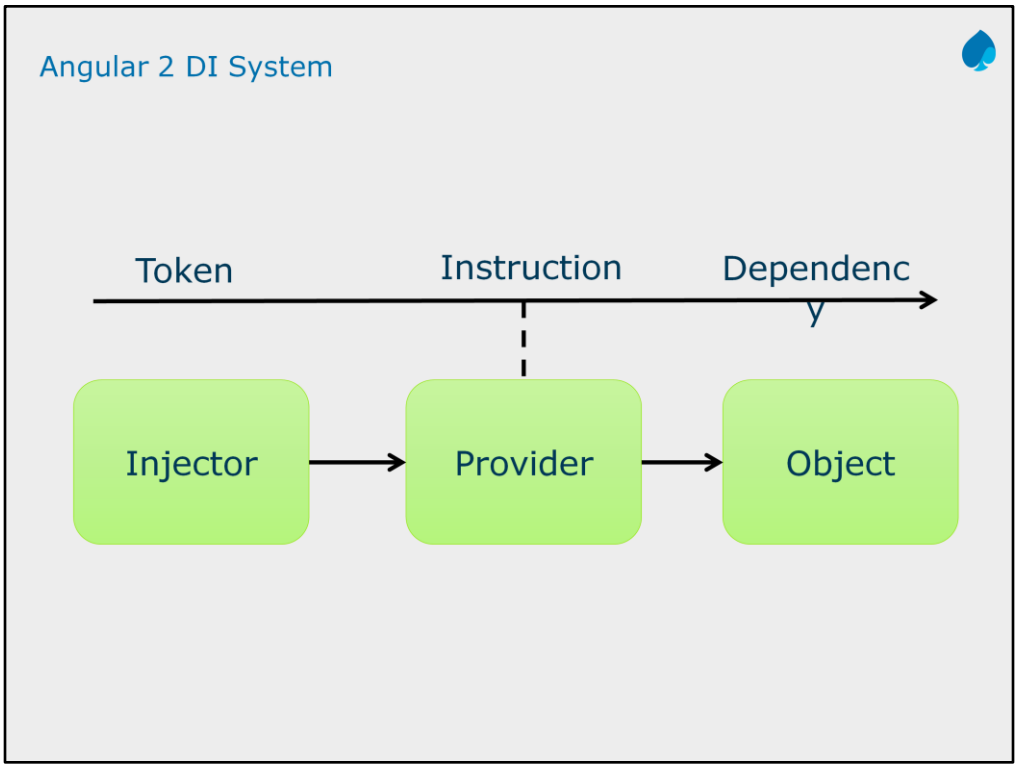
Add instructor notes here.



A provider is an instruction that describes how an object for a certain token is created.

Instructor Notes:

Add instructor notes here.



The DI in Angular 2 basically consists of three things:

Injector : The injector object that exposes APIs to create instances of dependencies.

Provider : A provider is like a recipe that tells the injector how to create an instance of a dependency. A provider takes a token and maps that to a factory function that creates an object.

Dependency : A dependency is the type of which an object should be created.

Instructor Notes:

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 depedency injection**
  - **constructor(private _exampleService: ExampleService) {}**

Instructor Notes:

Add instructor notes here.

Demo

DemoService

Add the notes here.

Instructor Notes:

Add instructor notes here.

Summary

- 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).
- ***@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.