SHREE H. N. SHUKLA COLLEGE OF I.T. & MGMT.

(AFFILIATED TO SAURASHTRA UNIVERSITY)

Lt. Shree Chimanbhai Shukla

MSCIT SEM-3 ANGULARJS

Shree H.N.Shukla college2
vaishali nagar
Near Amrapali Under Bridge,
Raiya road
Rajkot
Ph No:-0281 2440478

Shree H.N.Shukla college3
vaishali nagar
Near Amrapali Under Bridge,
Raiya road
Rajkot
Ph No:-0281 2440478

SHREE H. N. SHUKLA COLLEGE OF I.T. & MGMT.
(AFFILIATED TO SAURASHTRA UNIVERSITY)

Unit : 3
Dependency Injection and service in Angular

index

❖ (1)Understanding Dependency Injection (DI)

Dependency injection, or DI, is one of the fundamental concepts in Angular. DI is wired into the Angular framework and allows classes with Angular decorators, such as Components, Directives, Pipes, and Injectables, to configure dependencies that they need.

Two main roles exist in the DI system: dependency consumer and dependency provider.

Angular facilitates the interaction between dependency consumers and dependency providers using an abstraction called Injector. When a dependency is requested, the injector checks its registry to see if there is an instance already available there. If not, a new instance is created and stored in the registry. Angular creates an application-wide injector (also known as "root" injector) during the application bootstrap process, as well as any other injectors as needed. In most cases you don't need to manually create injectors, but you should know that there is a layer that connects providers and consumers.

This topic covers basic scenarios of how a class can act as a dependency. Angular also allows you to use functions, objects, primitive types such as string or Boolean, or any other types as dependencies. For more information, see Dependency providers.

Providing dependency

Imagine there is a class called HeroService that needs to act as a dependency in a component.

The first step is to add the @Injectable decorator to show that the class can be injected.

```
content_copy@Injectable()

class HeroService {}
```

The next step is to make it available in the DI by providing it. A dependency can be provided in multiple places:

- At the Component level, using the providers field of the @Component decorator. In this case the HeroService becomes available to all instances of this component and other components and directives used in the template. For example:

```
content_copy@Component({

  selector: 'hero-list',

  template: '...',

  providers: [HeroService]

})

class HeroListComponent {}
```

When you register a provider at the component level, you get a new instance of the service with each new instance of that component.

- At the NgModule level, using the providers field of the @NgModule decorator. In this scenario, the HeroService is available to all components, directives, and pipes declared in this NgModule or other NgModule which is within the same ModuleInjector applicable for this NgModule. When you register a provider with a specific NgModule, the same instance of a service is available to all applicable components, directives and pipes. To understand all edge-cases, see Hierarchical injectors. For example:

```
content_copy@NgModule({
  declarations: [HeroListComponent]
  providers: [HeroService]
})
class HeroListModule {}
```

- At the application root level, which allows injecting it into other classes in the application. This can be done by adding the providedIn: 'root' field to the @Injectable decorator:

```
content_copy@Injectable({
  providedIn: 'root'
})
class HeroService {}
```

When you provide the service at the root level, Angular creates a single, shared instance of the HeroService and injects it into any class that asks for it. Registering the provider in the @Injectable metadata also allows Angular to optimize an app by removing the service from the compiled application if it isn't used, a process known as tree-shaking.

Injecting a dependency
The most common way to inject a dependency is to declare it in a class constructor. When Angular creates a new instance of a component, directive, or pipe class, it determines which services or other dependencies that class needs by looking at the constructor parameter types. For example, if the HeroListComponent needs the HeroService, the constructor can look like this:

```
content_copy@Component({ … })
```

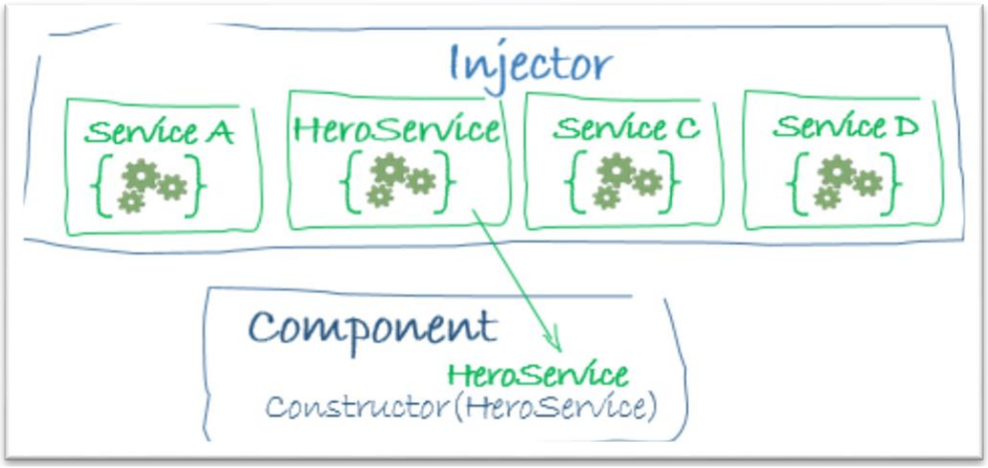
```
class HeroListComponent {
  constructor(private service: HeroService) {}
}
```

Another option is to use the inject method:

```
content_copy@Component({ … })
class HeroListComponent {
  private service = inject(HeroService);
}
```

When Angular discovers that a component depends on a service, it first checks if the injector has any existing instances of that service. If a requested service instance doesn't yet exist, the injector creates one using the registered provider, and adds it to the injector before returning the service to Angular.

When all requested services have been resolved and returned, Angular can call the component's constructor with those services as arguments.



❖(2)Services

AngularJS services are substitutable objects that are wired together using dependency injection (DI). You can use services to organize and share code across your app.

AngularJS services are:

- Lazily instantiated – AngularJS only instantiates a service when an application component depends on it.
- Singletons – Each component dependent on a service gets a reference to the single instance generated by the service factory.

AngularJS offers several useful services (like $http), but for most applications you'll also want to create your own.

❖ Using a Service

To use an AngularJS service, you add it as a dependency for the component (controller, service, filter or directive) that depends on the service. AngularJS's dependency injection subsystem takes care of the rest.

Edit in Plunker

index.htmlscript.jsprotractor.js

```
<div id="simple" ng-controller="MyController">
  <p>Let's try this simple notify service, injected into the controller...</p>
  <input ng-init="message='test'" ng-model="message" >
  <button ng-click="callNotify(message);">NOTIFY</button>
  <p>(you have to click 3 times to see an alert)</p>
</div>
```

❖ (3)Creating Services

Application developers are free to define their own services by registering the service's name and **service factory function**, with an AngularJS module.

The **service factory function** generates the single object or function that represents the service to the rest of the application. The object or function returned by the service is injected into any component (controller, service, filter or directive) that specifies a dependency on the service.

➢ Registering Services

Services are registered to modules via the Module API. Typically you use the Module factory API to register a service:

```
var myModule = angular.module('myModule', []);
myModule.factory('serviceId', function() {
  var shinyNewServiceInstance;
  // factory function body that constructs shinyNewServiceInstance
  return shinyNewServiceInstance;
});
```

Note that you are not registering a **service instance**, but rather a **factory function** that will create this instance when called.

➢ Dependencies

Services can have their own dependencies. Just like declaring dependencies in a controller, you declare dependencies by specifying them in the service's factory function signature.

For more on dependencies, see the dependency injection docs.

The example module below has two services, each with various dependencies:

```
var batchModule = angular.module('batchModule', []);

/**
 * The `batchLog` service allows for messages to be queued in memory and flushed
 * to the console.log every 50 seconds.
 *
 * @param {*} message Message to be logged.
 */
batchModule.factory('batchLog', ['$interval', '$log', function($interval, $log) {
  var messageQueue = [];

  function log() {
    if (messageQueue.length) {
      $log.log('batchLog messages: ', messageQueue);
      messageQueue = [];
    }
  }

  // start periodic checking
  $interval(log, 50000);

  return function(message) {
    messageQueue.push(message);
  }
}]);


/**
 * `routeTemplateMonitor` monitors each `$route` change and logs the current
 * template via the `batchLog` service.
 */
batchModule.factory('routeTemplateMonitor', ['$route', 'batchLog', '$rootScope',
  function($route, batchLog, $rootScope) {
    return {
      startMonitoring: function() {
        $rootScope.$on('$routeChangeSuccess', function() {
          batchLog($route.current ? $route.current.template : null);
        });
      }
    };
  }]);
```

In the example, note that:

- The batchLog service depends on the built-in $interval and $log services.
- The routeTemplateMonitor service depends on the built-in $route service and our custom batchLog service.
- Both services use the array notation to declare their dependencies.
- The order of identifiers in the array is the same as the order of argument names in the factory function.

➢ Registering a Service with $provide

You can also register services via the $provide service inside of a module's config function:

```
angular.module('myModule', []).config(['$provide', function($provide) {
  $provide.factory('serviceId', function() {
    var shinyNewServiceInstance;
    // factory function body that constructs shinyNewServiceInstance
    return shinyNewServiceInstance;
  });
}]);
```

This technique is often used in unit tests to mock out a service's dependencies.

➢ Unit Testing

The following is a unit test for the notify service from the Creating AngularJS Services example above. The unit test example uses a Jasmine spy (mock) instead of a real browser alert.

```
var mock, notify;
beforeEach(module('myServiceModule'));
beforeEach(function() {
  mock = {alert: jasmine.createSpy()};

  module(function($provide) {
    $provide.value('$window', mock);
  });
  inject(function($injector) {
    notify = $injector.get('notify');
  });
});

it('should not alert first two notifications', function() {
  notify('one');
  notify('two');

  expect(mock.alert).not.toHaveBeenCalled();
});

it('should alert all after third notification', function() {
  notify('one');
  notify('two');
  notify('three');

  expect(mock.alert).toHaveBeenCalledWith("one\ntwo\nthree");
});

it('should clear messages after alert', function() {
  notify('one');
  notify('two');
  notify('third');
  notify('more');
  notify('two');
  notify('third');

  expect(mock.alert.calls.count()).toEqual(2);
  expect(mock.alert.calls.mostRecent().args).toEqual(["more\ntwo\nthird"]);
});
```

➢ Introduction

In this article, we are going to explore the steps needed to create services in Angular applications along with the concept of dependency injection.

What is dependecy Injection and why do we use it ?

Dependency Injection (DI) is a mechanism where the required resources will be injected into the code automatically. Angular comes with a in-built dependency injection subsystem.

- DI allows developers to reuse the code across application.
- DI makes the application development and testing much easier.
- DI makes the code loosely coupled.
- DI allows the developer to ask for the dependencies from Angular. There is no need for the developer to explicitly create/instantiate them.

What is Service and why do we use it?

- A service in Angular is a class which contains some functionality that can be reused across the application. A service is a singleton object. Angular services are a mechanism of abstracting shared code and functionality throughout the application.
- Angular Services come as objects which are wired together using dependency injection.
- Angular provides a few inbuilt services. We can also create custom services.

Why Services?

- Services can be used to share the code across components of an application.
- Services can be used to make HTTP requests.

Creating a Service

Create a service class using the following command.

```
ng generate service Article
```

The above command will create a service class (article.service.ts) as shown below.

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

@Injectable({
  providedIn: 'root'
})
export class ArticleService {

  constructor() { }
}
```

@Injectable() decorator makes the class injectable into application components.

Providing a Service

Services can be provided in an Angular applications in any of the following ways:

The first way to register service is to specify providedIn property using @Injectable decorator. This property is added by default when you generate a service using Angular CLI.

```
1. import { Injectable } from '@angular/core';
2.
3. @Injectable({
4.   providedIn: 'root'
5. })
6. export class ArticleService {
7.
8.   constructor() { }
9. }
```

Line 4: providedIn property registers articleService at the root level (app module).

When the ArticleService is provided at the root level, Angular creates a singleton instance of the service class and injects the same instance into any class that uses this service class. In addition, Angular also optimizes the application if registered through providedIn property by removing the service class if none of the components use it.

There is also a way to limit the scope of the service class by registering it in the providers' property inside @Component decorator. Providers in component decorator and module decorator are independent. Providing a service class inside component creates a separate instance for that component and its nested components.

Add the below code in app.components.ts,

```
import { Component } from '@angular/core';
import { ArticleService } from './article.service';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  providers : [ArticleService]
})
export class AppComponent {
  title = 'FormsProject';
}
```

Services can also be provided across the application by registering it using providers property in @Ngmodule decorator of any module.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { ReactiveFormsModule } from '@angular/forms';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import {Form, FormsModule} from '@angular/forms';
import { ArticleFormComponent } from './article-form/article-form.component';
import { RegistrationFormComponent } from './registration-form/registration-form.component';
import { ArticleService } from './article.service';
@NgModule({
```

```
11.   declarations: [
12.     AppComponent,
13.     ArticleFormComponent,
14.     RegistrationFormComponent
15.   ],
16.   imports: [
17.     BrowserModule,
18.     AppRoutingModule,
19.     FormsModule,
20.     ReactiveFormsModule
21.   ],
22.   providers: [ArticleService],
23.   bootstrap: [AppComponent]
24. })
25. export class AppModule { }
```

Line 22: When the service class is added in the providers property of the root module, all the directives and components will have access to the same instance of the service.

Injecting a Service

The only way to inject a service into a component/directive or any other class is through a constructor. Add a constructor in a component class with service class as an argument as shown below,

Here, ArticleService will be injected into the component through constructor injection by the framework.

```
import { Component } from '@angular/core';
import { ArticleService } from './article.service';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  providers : [ArticleService]
})
export class AppComponent {
  title = 'FormsProject';

  constructor(private articleService: ArticleService){ }
}
```

Problem Statement

Create an Article Component which fetches article details like id, name and displays them on the page in a list format. Store the article details in an array and fetch the data using a custom service.

Demosteps

Create ArticleComponent by using the following CLI command

```
ng generate component Article
```

Create a file with name Article.ts under book folder and add the following code.

```
export class Article {
    id: number;
    name: string;
}
```

Create a file with name Article-data.ts under book folder and add the following code.

```
import {Article} from './Article';
export var ARTICLES: Article[] = [
    { "id": 1, "name": "Angular Basic" },
    { "id": 2, "name": "Template in Angular" },
    { "id": 3, "name": "Nested component" },
    { "id": 4, "name": "Reactive component" },
    { "id": 5, "name": "Change detection technique" }
];
```

Create a service called ArticleService under book folder using the following CLI command,

```
ng generate service Article
```

Add the following code in **a**rticle.service.ts

```
import { Injectable } from '@angular/core';
import {ARTICLES} from './Article-data';
import {Article} from './Article';

@Injectable({
  providedIn: 'root'
})
export class ArticleService {

getArticles ()
{
  return ARTICLES;
}
}
```

Add the following code in article.component.ts file

```
import { Component, OnInit } from '@angular/core';
import {ArticleService} from './article.service';
import { Article } from './Article';
@Component({
  selector: 'app-article',
  templateUrl: './article.component.html',
  styleUrls: ['./article.component.css']
})
export class ArticleComponent implements OnInit {
articles : Article[];

  constructor(private articelService : ArticleService) { }
  getArticles()
  {
    this.articles=this.articelService.getArticles()
  }
  ngOnInit() {
    this.getArticles()
  }
}
```

Write the below-given code in article.component.html

```
<h2>My Articles</h2>
<ul class="Articles">
  <li *ngFor="let article of Articles">
    <span class="badge">{{article.id}}</span> {{article.name}}
  </li>
</ul>
```

Add the following code in article.component.css which has styles for books.

```
.Articles {
    margin: 0 0 2em 0;
    list-style-type: none;
    padding: 0;
    width: 15em;
}
.Articles li {
    cursor: pointer;
    position: relative;
    left: 0;
    background-color: #EEE;
    margin: .5em;
    padding: .3em 0;
    height: 1.6em;
    border-radius: 4px;
}
.Articles li:hover {
    color: #607D8B;
    background-color: #DDD;
    left: .1em;
}
.Articles .badge {
    display: inline-block;
    font-size: small;
    color: white;
    padding: 0.8em 0.7em 0 0.7em;
    background-color: #607D8B;
    line-height: 1em;
    position: relative;
    left: -1px;
    top: -4px;
    height: 1.8em;
    margin-right: .8em;
    border-radius: 4px 0 0 4px;
}
```

Add the following code in app.component.html.
<app-article></app-article>

❖(4)Injecting the service into components

Service is a special class in Angular that is primarily used for inter-component communication. It is a class having a narrow & well-defined purpose that should perform a specific task. The function, any value, or any feature which may application required, are encompassed by the Service. In other words, sometimes, there are components that need a common pool to interact with each other mostly for data or information procurement, & Service makes it possible. The two (or more) components may or may not be related to each other. That means there may exist a parent-child relationship or nothing at all.
Basically, Service helps to organize & share the business logic, data model & functions with various components in the application, & it gets instantiated only once in the lifecycle of the application. For this, Services is written only once & can be injected into the different components, which use that particular Services.

Services and other dependencies are injected directly into the constructor of the component like this:

```
constructor(private _myService: MyService) {

}
```

By doing this, we are actually creating an instance of the service, which means we have to access all the public variables and methods of the service.

Syntax: To create a new service, we can use the below command:

```
// Generate service
ng generate sservice my-custom-service
```

Injecting a service into a component is pretty straightforward. For instance, suppose we have a service called *MyCustomService*. This is how we can inject it into a component:

- **MyCustomComponent.ts**

- Javascript

```
import {... } from "@angular/core";

import { MyCustomService } from "../...PATH";



@Component({

   selector: "...",

   templateUrl: "...",

   styleUrls: ["..."],

})

export class MyCustomComponent {



   // INJECTING SERVICE INTO THE CONSTRUCTOR

   constructor(private _myCustomService: MyCustomService) { }
```

```
    // USING THE SERVICE MEMBERS

    this._myCustomService.sampleMethod();

}
```

Before we proceed to inject the Service into the Component, we need to set up as many Components, as required. Please refer to the Components in Angular article for the detailed installation of the components in the application. Now, we have everything that we need. Since this demo is particularly for service injection.
This may not make any sense until and unless we get our hands dirty. So let's quickly create a service and see how it is injected and can be accessed easily. For this, we will create two simple custom components, let's say, Ladies and Gentlemen. There is no parent-child relationship between these two components. Both are absolutely independent. Gentlemen will greet Ladies with "Good Morning" with the click of a button. For this, we will use a service that will interact between the two components. We will call it *InteractionService*. First thing first, we will create our 2 components and 1 service. We will now create the first component using the following command:

```
ng generate component gentlemen
```

Quickly create our last component:

```
ng generate component ladies
```

Project Structure: The following structure will appear after completing the installation procedure:

Example: This example describes the injection of Service in the angular 6 components.

- **interaction.service.ts**

- Javascript

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


@Injectable({
    providedIn: 'root'
})
export class InteractionService {
    private _messageSource = new Subject<string>();


    greeting$ = this._messageSource.asObservable();


    sendMessage(message: string) {
        this._messageSource.next(message);
    }
}
```

This has been done. We will now inject this service into both our components.

- **gentlemen.component.html**

- HTML

```
<h1>GeeksforGeeks</h1>

<h3>

     Injecting the Service in Angular 6 Component

</h3>

<p>I am a Gentleman</p>

<button (click)="greetLadies()">

     Greet1

</button>

<button (click)="greetLadies1()">

     Greet2

</button>
```

- **gentlemen.component.ts**

- Javascript

```
import { Component, OnInit } from '@angular/core';

import { InteractionService }

    from "../interaction.service";
```

```
@Component({
    selector: 'app-gentlemen',
    templateUrl: './gentlemen.component.html',
    styleUrls: ['./gentlemen.component.css']
})
export class GentlemenComponent {
    // SERVICE INJECTION
    constructor(private _interactionService: InteractionService) { }


    greetLadies() {
        this._interactionService.sendMessage("Good Morning");
    }
    greetLadies1() {
        this._interactionService.sendMessage("Good Evening");
    }
}
```

- **gentleman.component.css**

- CSS

```css
h1 {
    color: green;
}
```

- **ladies.component.ts:**

- Javascript

```javascript
import { Component, OnInit } from '@angular/core';
import { InteractionService } from "../interaction.service";


@Component({
    selector: 'app-ladies',
    templateUrl: './ladies.component.html',
    styleUrls: ['./ladies.component.css']
})
```

```
export class LadiesComponent implements OnInit {

    // SERVICE INJECTION

    constructor(private _interactionService: InteractionService) { }



    ngOnInit() {

        this._interactionService.greeting$.subscribe(message => {

            console.log(message);

        })

    }

}
```

- **app.component.html**

- HTML

```
<app-gentlemen></app-gentlemen>

<app-ladies></app-ladies>
```

- **app.component.ts**

- Javascript

```
import { Component } from '@angular/core';


@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'MyProject';
}
```

- **app.module.ts**

- Javascript

```
import { NgModule } from '@angular/core';
import { BrowserModule }
    from '@angular/platform-browser';
import { AppComponent } from './app.component';
import { GentlemenComponent }

    from './gentlemen/gentlemen.component';

import { LadiesComponent }

    from './ladies/ladies.component';



@NgModule({

    declarations: [

        AppComponent,

        GentlemenComponent,

        LadiesComponent

    ],

    imports: [

        BrowserModule

    ],

    providers: [],

    bootstrap: [AppComponent]
})

export class AppModule { }
```

This is how we can inject and use the service to interact between components. We just saw a use case of service injection.

❖(5)Understanding Dependency Hierarchical Injector

Hierarchical injectors

Injectors in Angular have rules that you can leverage to achieve the desired visibility of injectables in your applications. By understanding these rules, you can determine in which NgModule, Component, or Directive you should declare a provider.

NOTE:
This topic uses the following pictographs.

HTML ENTITIES	PICTOGRAPHS
□	red hibiscus (□)
□	sunflower (□)
□	tulip (□)
□	fern (□)
□	maple leaf (□)
□	whale (□)
□	dog (□)
□	hedgehog (□)

The applications you build with Angular can become quite large, and one way to manage this complexity is to split up the application into many small well-encapsulated modules, that are by themselves split up into a well-defined tree of components.

There can be sections of your page that works in a completely independent way than the rest of the application, with its own local copies of the services and other dependencies that it needs. Some of the services that these sections of the application use might be shared with other parts of the application, or with parent components that are further up in the component tree, while other dependencies are meant to be private.

With hierarchical dependency injection, you can isolate sections of the application and give them their own private dependencies not shared with the rest of the application, or have parent components share certain dependencies with its child components only but not with the rest of the component tree, and so on. Hierarchical dependency injection enables you to share dependencies between different parts of the application only when and if you need to.

Types of injector hierarchies

Injectors in Angular have rules that you can leverage to achieve the desired visibility of injectables in your applications. By understanding these rules, you can determine in which NgModule, Component, or Directive you should declare a provider.

Angular has two injector hierarchies:

INJECTOR HIERARCHIES	DETAILS
ModuleInjector hierarchy	Configure a ModuleInjector in this hierarchy using an @NgModule() or @Injectable() annotation.
ElementInjector hierarchy	Created implicitly at each DOM element. An ElementInjector is empty by default unless you configure it in the providers property on @Directive() or @Component().

ModuleInjector

The ModuleInjector can be configured in one of two ways by using:

- The @Injectable() providedIn property to refer to root or platform
- The @NgModule() providers array

TREE-SHAKING AND @INJECTABLE()

Using the @Injectable() providedIn property is preferable to using the @NgModule() providers array. With @Injectable() providedIn, optimization tools can perform tree-shaking, which removes services that your application isn't using. This results in smaller bundle sizes.

Tree-shaking is especially useful for a library because the application which uses the library may not have a need to inject it. Read more about tree-shakable providers in Introduction to services and dependency injection.

ModuleInjector is configured by the @NgModule.providers and NgModule.imports property. ModuleInjector is a flattening of all the providers arrays that can be reached by following the NgModule.imports recursively.

Child ModuleInjector hierarchies are created when lazy loading other @NgModules.

Provide services with the providedIn property of @Injectable() as follows:

```
content_copyimport { Injectable } from '@angular/core';


@Injectable({
  providedIn: 'root'  // <--provides this service in the root ModuleInjector
})
export class ItemService {
  name = 'telephone';
}
```

The @Injectable() decorator identifies a service class. The providedIn property configures a specific ModuleInjector, here root, which makes the service available in the root ModuleInjector.

Platform injector
There are two more injectors above root, an additional ModuleInjector and NullInjector().

Consider how Angular bootstraps the application with the following in main.ts:

```
content_copyplatformBrowserDynamic().bootstrapModule(AppModule).then(ref
=> {...})
```

The bootstrapModule() method creates a child injector of the platform injector which is configured by the AppModule. This is the root ModuleInjector.

The platformBrowserDynamic() method creates an injector configured by a PlatformModule, which contains platform-specific dependencies. This allows multiple applications to share a platform configuration. For example, a browser has only one URL bar, no matter how many applications you have running. You can configure additional platform-specific providers at the platform level by supplying extraProviders using the platformBrowser() function.

The next parent injector in the hierarchy is the NullInjector(), which is the top of the tree. If you've gone so far up the tree that you are looking for a service in the NullInjector(), you'll get an error unless you've used @Optional() because ultimately, everything ends at the NullInjector() and it returns an error or, in the case of @Optional(), null. For more information on @Optional(), see the @Optional() section of this guide.

The following diagram represents the relationship between the root ModuleInjector and its parent injectors as the previous paragraphs describe.

While the name root is a special alias, other ModuleInjector hierarchies don't have aliases. You have the option to create ModuleInjector hierarchies whenever a dynamically loaded component is created, such as with the Router, which will create child ModuleInjector hierarchies.

All requests forward up to the root injector, whether you configured it with the bootstrapModule() method, or registered all providers with root in their own services.

@INJECTABLE() VS. @NGMODULE()

If you configure an app-wide provider in the @NgModule() of AppModule, it overrides one configured for root in the @Injectable() metadata. You can do this to configure a non-default provider of a service that is shared with multiple applications.

Here is an example of the case where the component router configuration includes a non-default location strategy by listing its provider in the providers list of the AppModule.

SRC/APP/APP.MODULE.TS (PROVIDERS)

```
  content_copyproviders: [

    { provide: LocationStrategy, useClass: HashLocationStrategy }

  ]
```

ElementInjector

Angular creates ElementInjector hierarchies implicitly for each DOM element.

Providing a service in the @Component() decorator using its providers or viewProviders property configures an ElementInjector. For example, the following TestComponent configures the ElementInjector by providing the service as follows:

```
content_copy@Component({
  …
  providers: [{ provide: ItemService, useValue: { name: 'lamp' } }]
})
export class TestComponent
```

NOTE:
See the resolution rules section to understand the relationship between the ModuleInjector tree and the ElementInjector tree.

When you provide services in a component, that service is available by way of the ElementInjector at that component instance. It may also be visible at child component/directives based on visibility rules described in the resolution rules section.

When the component instance is destroyed, so is that service instance.

@Directive() and @Component()
A component is a special type of directive, which means that just as @Directive() has a providers property, @Component() does too. This means that directives as well as components can configure providers, using the providers property. When you configure a provider for a component or directive using the providers property, that provider belongs to the ElementInjector of that component or directive. Components and directives on the same element share an injector.

Resolution rules
When resolving a token for a component/directive, Angular resolves it in two phases:

1. Against its parents in the ElementInjector hierarchy.
2. Against its parents in the ModuleInjector hierarchy.

When a component declares a dependency, Angular tries to satisfy that dependency with its own ElementInjector. If the component's injector lacks the provider, it passes the request up to its parent component's ElementInjector.

The requests keep forwarding up until Angular finds an injector that can handle the request or runs out of ancestor ElementInjector hierarchies.

If Angular doesn't find the provider in any ElementInjector hierarchies, it goes back to the element where the request originated and looks in the ModuleInjector hierarchy. If Angular still doesn't find the provider, it throws an error.

If you have registered a provider for the same DI token at different levels, the first one Angular encounters is the one it uses to resolve the dependency. If, for example, a provider is registered locally in the component that needs a service, Angular doesn't look for another provider of the same service.

Resolution modifiers

Angular's resolution behavior can be modified with @Optional(), @Self(), @SkipSelf() and @Host(). Import each of them from @angular/core and use each in the component class constructor when you inject your service.

For a working application showcasing the resolution modifiers that this section covers, see the resolution modifiers example / download example.

Types of modifiers

Resolution modifiers fall into three categories:

- What to do if Angular doesn't find what you're looking for, that is @Optional()
- Where to start looking, that is @SkipSelf()
- Where to stop looking, @Host() and @Self()

By default, Angular always starts at the current Injector and keeps searching all the way up. Modifiers allow you to change the starting, or *self*, location and the ending location.

Additionally, you can combine all of the modifiers except @Host() and @Self() and of course @SkipSelf() and @Self().

@Optional()

@Optional() allows Angular to consider a service you inject to be optional. This way, if it can't be resolved at runtime, Angular resolves the service as null, rather than throwing an error. In the following example, the service, OptionalService, isn't provided in the service, @NgModule(), or component class, so it isn't available anywhere in the app.

src/app/optional/optional.component.ts

```
content_copyexport class OptionalComponent {

  constructor(@Optional() public optional?: OptionalService) {}

}
```

@Self()

Use @Self() so that Angular will only look at the ElementInjector for the current component or directive.

A good use case for @Self() is to inject a service but only if it is available on the current host element. To avoid errors in this situation, combine @Self() with @Optional().

For example, in the following SelfComponent, notice the injected LeafService in the constructor.

src/app/self-no-data/self-no-data.component.ts

```
content_copy@Component({

  selector: 'app-self-no-data',

  templateUrl: './self-no-data.component.html',

  styleUrls: ['./self-no-data.component.css']

})

export class SelfNoDataComponent {

  constructor(@Self() @Optional() public leaf?: LeafService) { }

}
```

In this example, there is a parent provider and injecting the service will return the value, however, injecting the service with @Self() and @Optional() will return null because @Self() tells the injector to only search in the current host element.

Another example shows the component class with a provider for FlowerService. In this case, the injector looks no further than the current ElementInjector because it finds the FlowerService and returns the tulip □.

src/app/self/self.component.ts

```
content_copy@Component({

  selector: 'app-self',

  templateUrl: './self.component.html',

  styleUrls: ['./self.component.css'],

  providers: [{ provide: FlowerService, useValue: { emoji: '□' } }]


})

export class SelfComponent {

  constructor(@Self() public flower: FlowerService) {}

}
```

@SkipSelf()

@SkipSelf() is the opposite of @Self(). With @SkipSelf(), Angular starts its search for a service in the parent ElementInjector, rather than in the current one. So if the parent ElementInjector were using the fern □ value for emoji, but you had maple leaf □ in the component's providers array, Angular would ignore maple leaf □ and use fern □.

To see this in code, assume that the following value for emoji is what the parent component were using, as in this service:

src/app/leaf.service.ts

```
content_copyexport class LeafService {

  emoji = '□';
}
```

Imagine that in the child component, you had a different value, maple leaf □ but you wanted to use the parent's value instead. This is when you'd use @SkipSelf():

src/app/skipself/skipself.component.ts

```
content_copy@Component({
  selector: 'app-skipself',
  templateUrl: './skipself.component.html',
  styleUrls: ['./skipself.component.css'],
  // Angular would ignore this LeafService instance
  providers: [{ provide: LeafService, useValue: { emoji: '□' } }]
})
export class SkipselfComponent {
  // Use @SkipSelf() in the constructor
  constructor(@SkipSelf() public leaf: LeafService) { }
}
```

In this case, the value you'd get for emoji would be fern □, not maple leaf □.

@SkipSelf() with @Optional()

Use @SkipSelf() with @Optional() to prevent an error if the value is null. In the following example, the Person service is injected in the constructor. @SkipSelf() tells Angular to skip the current injector and @Optional() will prevent an error should the Person service be null.

```
content_copyclass Person {
  constructor(@Optional() @SkipSelf() parent?: Person) {}
}
```

@Host()

@Host() lets you designate a component as the last stop in the injector tree when searching for providers. Even if there is a service instance further up the tree, Angular won't continue looking. Use @Host() as follows:

src/app/host/host.component.ts

```
content_copy@Component({

  selector: 'app-host',

  templateUrl: './host.component.html',

  styleUrls: ['./host.component.css'],

  //  provide the service

  providers: [{ provide: FlowerService, useValue: { emoji: '□' } }]

})

export class HostComponent {

  // use @Host() in the constructor when injecting the service

  constructor(@Host() @Optional() public flower?: FlowerService) { }


}
```

Since HostComponent has @Host() in its constructor, no matter what the parent of HostComponent might have as a flower.emoji value, the HostComponent will use tulip □.

Logical structure of the template

When you provide services in the component class, services are visible within the ElementInjector tree relative to where and how you provide those services.

Understanding the underlying logical structure of the Angular template will give you a foundation for configuring services and in turn control their visibility.

Components are used in your templates, as in the following example:

```
content_copy<app-root>

  <app-child></app-child>

</app-root>
```

NOTE:
Usually, you declare the components and their templates in separate files. For the purposes of understanding how the injection system works, it is useful to look at them from the point of view of a combined logical tree. The term *logical* distinguishes it from the render tree, which is your application's DOM tree. To mark the locations of where the component templates are located, this guide uses the <#VIEW> pseudo-element, which doesn't actually exist in the render tree and is present for mental model purposes only.

The following is an example of how the <app-root> and <app-child> view trees are combined into a single logical tree:

```
content_copy<app-root>

  <#VIEW>

    <app-child>

     <#VIEW>

       …content goes here…

     </#VIEW>

    </app-child>

  </#VIEW>

</app-root>
```

Understanding the idea of the <#VIEW> demarcation is especially significant when you configure services in the component class.

Providing services in @Component()

How you provide services using a @Component() (or @Directive()) decorator determines their visibility. The following sections demonstrate providers and viewProviders along with ways to modify service visibility with @SkipSelf() and @Host().

A component class can provide services in two ways:

ARRAYS	DETAILS
With a providers array	content_copy@Component({ … providers: [{provide: FlowerService, useValue: {emoji: '□'}}] })
With a viewProviders array	content_copy@Component({ … viewProviders: [{provide: AnimalService, useValue: {emoji: '□'}}] })

To understand how the providers and viewProviders influence service visibility differently, the following sections build a live example / download example step-by-step and compare the use of providers and viewProviders in code and a logical tree.

NOTE:
In the logical tree, you'll see @Provide, @Inject, and @NgModule, which are not real HTML attributes but are here to demonstrate what is going on under the hood.

ANGULAR SERVICE ATTRIBUTE	DETAILS
@Inject(Token)=>Value	Demonstrates that if Token is injected at this location in the logical tree its value would be Value.
@Provide(Token=Value)	Demonstrates that there is a declaration of Token provider with value Value at this location in the logical tree.
@NgModule(Token)	Demonstrates that a fallback NgModule injector should be used at this location.

Example app structure
The example application has a FlowerService provided in root with an emoji value of red hibiscus □.

src/app/flower.service.ts
```
content_copy@Injectable({

  providedIn: 'root'

})

export class FlowerService {

  emoji = '□';

}
```

Consider an application with only an AppComponent and a ChildComponent. The most basic rendered view would look like nested HTML elements such as the following:

```
content_copy<app-root> <!-- AppComponent selector -->
  <app-child> <!-- ChildComponent selector -->
  </app-child>
</app-root>
```

However, behind the scenes, Angular uses a logical view representation as follows when resolving injection requests:

```
content_copy<app-root> <!-- AppComponent selector -->
  <#VIEW>
    <app-child> <!-- ChildComponent selector -->
      <#VIEW>
      </#VIEW>
    </app-child>
  </#VIEW>
</app-root>
```

The <#VIEW> here represents an instance of a template. Notice that each component has its own <#VIEW>.

Knowledge of this structure can inform how you provide and inject your services, and give you complete control of service visibility.

Now, consider that <app-root> injects the FlowerService:

src/app/app.component.ts

```
content_copyexport class AppComponent  {
  constructor(public flower: FlowerService) {}
}
```

Add a binding to the <app-root> template to visualize the result:

src/app/app.component.html

```
    content_copy<p>Emoji from FlowerService: {{flower.emoji}}</p>
```

The output in the view would be:

```
    Emoji from FlowerService: □
```

In the logical tree, this would be represented as follows:

```
    content_copy<app-root @NgModule(AppModule)

            @Inject(FlowerService) flower=>"□">

      <#VIEW>

        <p>Emoji from FlowerService: {{flower.emoji}} (□)</p>

        <app-child>

          <#VIEW>

          </#VIEW>

        </app-child>

      </#VIEW>

    </app-root>
```

When <app-root> requests the FlowerService, it is the injector's job to resolve the FlowerService token. The resolution of the token happens in two phases:

1. The injector determines the starting location in the logical tree and an ending location of the search. The injector begins with the starting location and looks for the token at each level in the logical tree. If the token is found it is returned.
2. If the token is not found, the injector looks for the closest parent @NgModule() to delegate the request to.

In the example case, the constraints are:

1. Start with <#VIEW> belonging to <app-root> and end with <app-root>.
   - Normally the starting point for search is at the point of injection. However, in this case <app-root> @Components are special in that they also include their own viewProviders, which is why the search starts at <#VIEW> belonging to <app-root>. This would not be the case for a directive matched at the same location.
   - The ending location happens to be the same as the component itself, because it is the topmost component in this application.
2. The AppModule acts as the fallback injector when the injection token can't be found in the ElementInjector hierarchies.

Using the providers array

Now, in the ChildComponent class, add a provider for FlowerService to demonstrate more complex resolution rules in the upcoming sections:

src/app/child.component.ts

```
content_copy@Component({

  selector: 'app-child',

  templateUrl: './child.component.html',

  styleUrls: ['./child.component.css'],

  // use the providers array to provide a service

  providers: [{ provide: FlowerService, useValue: { emoji: '□' } }]

})


export class ChildComponent {

  // inject the service

  constructor( public flower: FlowerService) { }

}
```

Now that the FlowerService is provided in the @Component() decorator, when the <app-child> requests the service, the injector has only to look as far as the ElementInjector in the <app-child>. It won't have to continue the search any further through the injector tree.

The next step is to add a binding to the ChildComponent template.

src/app/child.component.html

```
    content_copy<p>Emoji from FlowerService: {{flower.emoji}}</p>
```

To render the new values, add <app-child> to the bottom of the AppComponent template so the view also displays the sunflower:

Child Component

Emoji from FlowerService: □

In the logical tree, this is represented as follows:

```
    content_copy<app-root @NgModule(AppModule)

          @Inject(FlowerService) flower=>"□">

      <#VIEW>

        <p>Emoji from FlowerService: {{flower.emoji}} (□)</p>

        <app-child @Provide(FlowerService="□")

                 @Inject(FlowerService)=>"□"> <!-- search ends here -->

          <#VIEW> <!-- search starts here -->

            <h2>Child Component</h2>

            <p>Emoji from FlowerService: {{flower.emoji}} (□)</p>

          </#VIEW>

        </app-child>
```

```
  </#VIEW>

</app-root>
```

When <app-child> requests the FlowerService, the injector begins its search at the <#VIEW> belonging to <app-child> (<#VIEW> is included because it is injected from @Component()) and ends with <app-child>. In this case, the FlowerService is resolved in the providers array with sunflower of the <app-child>. The injector doesn't have to look any further in the injector tree. It stops as soon as it finds the FlowerService and never sees the red hibiscus .

Using the viewProviders array
Use the viewProviders array as another way to provide services in the @Component() decorator. Using viewProviders makes services visible in the <#VIEW>.

The steps are the same as using the providers array, with the exception of using the viewProviders array instead.

For step-by-step instructions, continue with this section. If you can set it up on your own, skip ahead to Modifying service availability.

The example application features a second service, the AnimalService to demonstrate viewProviders.

First, create an AnimalService with an emoji property of whale :

src/app/animal.service.ts

```
content_copyimport { Injectable } from '@angular/core';


@Injectable({

  providedIn: 'root'

})

export class AnimalService {

  emoji = '';
}
```

Following the same pattern as with the FlowerService, inject the AnimalService in the AppComponent class:

src/app/app.component.ts

```
content_copyexport class AppComponent  {

  constructor(public flower: FlowerService, public animal: AnimalService) {}

}
```

NOTE:
You can leave all the FlowerService related code in place as it will allow a comparison with the AnimalService.

Add a viewProviders array and inject the AnimalService in the <app-child> class, too, but give emoji a different value. Here, it has a value of dog □.

src/app/child.component.ts

```
content_copy@Component({

  selector: 'app-child',

  templateUrl: './child.component.html',

  styleUrls: ['./child.component.css'],

  // provide services

  providers: [{ provide: FlowerService, useValue: { emoji: '□' } }],

  viewProviders: [{ provide: AnimalService, useValue: { emoji: '□' } }]

})


export class ChildComponent {

  // inject service
  constructor( public flower: FlowerService, public animal: AnimalService) { }

}
```

Add bindings to the ChildComponent and the AppComponent templates. In the ChildComponent template, add the following binding:

src/app/child.component.html

```
content_copy<p>Emoji from AnimalService: {{animal.emoji}}</p>
```

Additionally, add the same to the AppComponent template:

src/app/app.component.html

```
content_copy<p>Emoji from AnimalService: {{animal.emoji}}</p>
```

Now you should see both values in the browser:

```
AppComponent

Emoji from AnimalService: □



Child Component

Emoji from AnimalService: □
```

The logic tree for this example of viewProviders is as follows:

```
content_copy<app-root @NgModule(AppModule)

        @Inject(AnimalService) animal=>"□">

  <#VIEW>

    <app-child>

      <#VIEW @Provide(AnimalService="□")

           @Inject(AnimalService=>"□")>
       <!-- ^^using viewProviders means AnimalService is available in <#VIEW>--
>
       <p>Emoji from AnimalService: {{animal.emoji}} (□)</p>

      </#VIEW>

    </app-child>

  </#VIEW>

</app-root>
```

Just as with the FlowerService example, the AnimalService is provided in the <app-child> @Component() decorator. This means that since the injector first looks in the ElementInjector of the component, it finds the AnimalService value of dog □. It doesn't need to continue searching the ElementInjector tree, nor does it need to search the ModuleInjector.

providers vs. viewProviders
To see the difference between using providers and viewProviders, add another component to the example and call it InspectorComponent. InspectorComponent will be a child of the ChildComponent. In inspector.component.ts, inject the FlowerService and AnimalService in the constructor:

src/app/inspector/inspector.component.ts

```
content_copyexport class InspectorComponent {

  constructor(public flower: FlowerService, public animal: AnimalService) { }

}
```

You do not need a providers or viewProviders array. Next, in inspector.component.html, add the same markup from previous components:

src/app/inspector/inspector.component.html

```
content_copy<p>Emoji from FlowerService: {{flower.emoji}}</p>

<p>Emoji from AnimalService: {{animal.emoji}}</p>
```

Remember to add the InspectorComponent to the AppModule declarations array.

src/app/app.module.ts

```
content_copy@NgModule({
  imports:      [ BrowserModule, FormsModule ],
  declarations: [ AppComponent, ChildComponent, InspectorComponent ],
  bootstrap:    [ AppComponent ],
  providers: []
})
export class AppModule { }
```

Next, make sure your child.component.html contains the following:

src/app/child/child.component.html

```
content_copy<p>Emoji from FlowerService: {{flower.emoji}}</p>
<p>Emoji from AnimalService: {{animal.emoji}}</p>


<div class="container">
  <h3>Content projection</h3>
      <ng-content></ng-content>
</div>


<h3>Inside the view</h3>
<app-inspector></app-inspector>
```

The first two lines, with the bindings, are there from previous steps. The new parts are <ng-content> and <app-inspector>. <ng-content> allows you to project content, and <app-inspector> inside the ChildComponent template makes the InspectorComponent a child component of ChildComponent.

Next, add the following to app.component.html to take advantage of content projection.

src/app/app.component.html

```
content_copy<app-child><app-inspector></app-inspector></app-child>
```

The browser now renders the following, omitting the previous examples for brevity:

```
//…Omitting previous examples. The following applies to this section.


Content projection: this is coming from content. Doesn't get to see

puppy because the puppy is declared inside the view only.


Emoji from FlowerService: 

Emoji from AnimalService: 


Emoji from FlowerService: 

Emoji from AnimalService: 
```

These four bindings demonstrate the difference between providers and viewProviders. Since the dog is declared inside the <#VIEW>, it isn't visible to the projected content. Instead, the projected content sees the whale .

The next section though, where InspectorComponent is a child component of ChildComponent, InspectorComponent is inside the <#VIEW>, so when it asks for the AnimalService, it sees the dog .

The AnimalService in the logical tree would look like this:

```
content_copy<app-root @NgModule(AppModule)
        @Inject(AnimalService) animal=>"□">
  <#VIEW>
    <app-child>
      <#VIEW @Provide(AnimalService="□")
           @Inject(AnimalService=>"□")>
        <!-- ^^using viewProviders means AnimalService is available in <#VIEW>-
->
        <p>Emoji from AnimalService: {{animal.emoji}} (□)</p>

        <div class="container">
          <h3>Content projection</h3>
          <app-inspector @Inject(AnimalService) animal=>"□">
            <p>Emoji from AnimalService: {{animal.emoji}} (□)</p>
          </app-inspector>
        </div>

        <app-inspector>
          <#VIEW @Inject(AnimalService) animal=>"□">
            <p>Emoji from AnimalService: {{animal.emoji}} (□)</p>
          </#VIEW>
        </app-inspector>
```

```
          </#VIEW>

        </app-child>

      </#VIEW>

    </app-root>
```

The projected content of <app-inspector> sees the whale □, not the dog □, because the dog □ is inside the <app-child> <#VIEW>. The <app-inspector> can only see the dog □ if it is also within the <#VIEW>.

Modifying service visibility

This section describes how to limit the scope of the beginning and ending ElementInjector using the visibility decorators @Host(), @Self(), and @SkipSelf().

Visibility of provided tokens

Visibility decorators influence where the search for the injection token begins and ends in the logic tree. To do this, place visibility decorators at the point of injection, that is, the constructor(), rather than at a point of declaration.

To alter where the injector starts looking for FlowerService, add @SkipSelf() to the <app-child> @Inject declaration for the FlowerService. This declaration is in the <app-child> constructor as shown in child.component.ts:

```
content_copyconstructor(@SkipSelf() public flower : FlowerService) { }
```

With @SkipSelf(), the <app-child> injector doesn't look to itself for the FlowerService. Instead, the injector starts looking for the FlowerService at the ElementInjector or the <app-root>, where it finds nothing. Then, it goes back to the <app-child> ModuleInjector and finds the red hibiscus □ value, which is available because the <app-child> ModuleInjector and the <app-root> ModuleInjector are flattened into one ModuleInjector. Thus, the UI renders the following:

```
Emoji from FlowerService: □
```

In a logical tree, this same idea might look like this:

```
content_copy<app-root @NgModule(AppModule)
        @Inject(FlowerService) flower=>"□">
  <#VIEW>
    <app-child @Provide(FlowerService="□")>
      <#VIEW @Inject(FlowerService, SkipSelf)=>"□">
        <!-- With SkipSelf, the injector looks to the next injector up the tree -->
      </#VIEW>
    </app-child>
  </#VIEW>
</app-root>
```

Though <app-child> provides the sunflower □, the application renders the red hibiscus □ because @SkipSelf() causes the current injector to skip itself and look to its parent.

If you now add @Host() (in addition to the @SkipSelf()) to the @Inject of the FlowerService, the result will be null. This is because @Host() limits the upper bound of the search to the <#VIEW>. Here's the idea in the logical tree:

```
content_copy<app-root @NgModule(AppModule)
        @Inject(FlowerService) flower=>"□">
  <#VIEW> <!-- end search here with null-->
    <app-child @Provide(FlowerService="□")> <!-- start search here -->
      <#VIEW @Inject(FlowerService, @SkipSelf, @Host, @Optional)=>null>
      </#VIEW>
      </app-parent>
  </#VIEW>

</app-root>
```

Here, the services and their values are the same, but @Host() stops the injector from looking any further than the <#VIEW> for FlowerService, so it doesn't find it and returns null.

NOTE:
The example application uses @Optional() so the application does not throw an error, but the principles are the same.

@SkipSelf() and viewProviders
The <app-child> currently provides the AnimalService in the viewProviders array with the value of dog . Because the injector has only to look at the ElementInjector of the <app-child> for the AnimalService, it never sees the whale .

As in the FlowerService example, if you add @SkipSelf() to the constructor for the AnimalService, the injector won't look in the ElementInjector of the current <app-child> for the AnimalService.

```
content_copyexport class ChildComponent {


  // add @SkipSelf()

  constructor(@SkipSelf() public animal : AnimalService) { }


}
```

Instead, the injector will begin at the <app-root> ElementInjector. Remember that the <app-child> class provides the AnimalService in the viewProviders array with a value of dog :

```
content_copy@Component({

  selector: 'app-child',
  …
  viewProviders:
  [{ provide: AnimalService, useValue: { emoji: '□' } }]
})
```

The logical tree looks like this with @SkipSelf() in <app-child>:

```
content_copy<app-root @NgModule(AppModule)
        @Inject(AnimalService=>"□")>
  <#VIEW><!-- search begins here -->
    <app-child>
      <#VIEW @Provide(AnimalService="□")
          @Inject(AnimalService, SkipSelf=>"□")>
        <!--Add @SkipSelf -->
      </#VIEW>
    </app-child>
  </#VIEW>
</app-root>
```

With @SkipSelf() in the <app-child>, the injector begins its search for the AnimalService in the <app-root> ElementInjector and finds whale □.

@Host() and viewProviders
If you add @Host() to the constructor for AnimalService, the result is dog □ because the injector finds the AnimalService in the <app-child> <#VIEW>. Here is the viewProviders array in the <app-child> class and @Host() in the constructor:

```
content_copy@Component({
  selector: 'app-child',
  …
  viewProviders:
  [{ provide: AnimalService, useValue: { emoji: '□' } }]

})
export class ChildComponent {
  constructor(@Host() public animal : AnimalService) { }
}
```

@Host() causes the injector to look until it encounters the edge of the <#VIEW>.

```
content_copy<app-root @NgModule(AppModule)
        @Inject(AnimalService=>"□")>
  <#VIEW>
    <app-child>
      <#VIEW @Provide(AnimalService="□")
             @Inject(AnimalService, @Host=>"□")> <!-- @Host stops search here -->
      </#VIEW>
    </app-child>
  </#VIEW>
</app-root>
```

Add a viewProviders array with a third animal, hedgehog □, to the app.component.ts @Component() metadata:

```
content_copy@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: [ './app.component.css' ],
  viewProviders: [{ provide: AnimalService, useValue: { emoji: '□' } }]
})
```

Next, add @SkipSelf() along with @Host() to the constructor for the Animal Service in child.component.ts. Here are @Host() and @SkipSelf() in the <app-child> constructor:

```
content_copyexport class ChildComponent {

  constructor(
  @Host() @SkipSelf() public animal : AnimalService) { }

}
```

When @Host() and @SkipSelf() were applied to the FlowerService, which is in the providers array, the result was null because @SkipSelf() starts its search in the <app-child> injector, but @Host() stops searching at <#VIEW> —where there is no FlowerService In the logical tree, you can see that the FlowerService is visible in <app-child>, not its <#VIEW>.

However, the AnimalService, which is provided in the AppComponent viewProviders array, is visible.

The logical tree representation shows why this is:

```
content_copy<app-root @NgModule(AppModule)
        @Inject(AnimalService=>"")>
  <#VIEW @Provide(AnimalService="")
         @Inject(AnimalService, @Optional)=>"">
    <!-- ^^@SkipSelf() starts here,  @Host() stops here^^ -->
    <app-child>
      <#VIEW @Provide(AnimalService="")
             @Inject(AnimalService, @SkipSelf, @Host, @Optional)=>"">
               <!-- Add @SkipSelf ^^-->
      </#VIEW>
      </app-child>
  </#VIEW>
</app-root>
```

@SkipSelf(), causes the injector to start its search for the AnimalService at the <app-root>, not the <app-child>, where the request originates, and @Host() stops the search at the <app-root> <#VIEW>. Since AnimalService is provided by way of the viewProviders array, the injector finds hedgehog in the <#VIEW>.

ElementInjector use case examples
The ability to configure one or more providers at different levels opens up useful possibilities. For a look at the following scenarios in a working app, see the heroes use case examples / download example.

Scenario: service isolation
Architectural reasons may lead you to restrict access to a service to the application domain where it belongs. For example, the guide sample includes

a VillainsListComponent that displays a list of villains. It gets those villains from a VillainsService.

If you provided VillainsService in the root AppModule (where you registered the HeroesService), that would make the VillainsService visible everywhere in the application, including the *Hero* workflows. If you later modified the VillainsService, you could break something in a hero component somewhere.

Instead, you can provide the VillainsService in the providers metadata of the VillainsListComponent like this:

src/app/villains-list.component.ts (metadata)

```
content_copy@Component({

  selector: 'app-villains-list',

  templateUrl: './villains-list.component.html',

  providers: [ VillainsService ]

})
```

By providing VillainsService in the VillainsListComponent metadata and nowhere else, the service becomes available only in the VillainsListComponent and its subcomponent tree.

VillainService is a singleton with respect to VillainsListComponent because that is where it is declared. As long as VillainsListComponent does not get destroyed it will be the same instance of VillainService but if there are multiple instances of VillainsListComponent, then each instance of VillainsListComponent will have its own instance of VillainService.

Scenario: multiple edit sessions
Many applications allow users to work on several open tasks at the same time. For example, in a tax preparation application, the preparer could be working on several tax returns, switching from one to the other throughout the day.

To demonstrate that scenario, imagine an outer HeroListComponent that displays a list of super heroes.

To open a hero's tax return, the preparer clicks on a hero name, which opens a component for editing that return. Each selected hero tax return opens in its own component and multiple returns can be open at the same time.

Each tax return component has the following characteristics:

- Is its own tax return editing session
- Can change a tax return without affecting a return in another component
- Has the ability to save the changes to its tax return or cancel them

Suppose that the HeroTaxReturnComponent had logic to manage and restore changes. That would be a straightforward task for a hero tax return. In the real world, with a rich tax return data model, the change management would be tricky. You could delegate that management to a helper service, as this example does.

The HeroTaxReturnService caches a single HeroTaxReturn, tracks changes to that return, and can save or restore it. It also delegates to the application-wide singleton HeroService, which it gets by injection.

src/app/hero-tax-return.service.ts

```
content_copyimport { Injectable } from '@angular/core';

import { HeroTaxReturn } from './hero';

import { HeroesService } from './heroes.service';


@Injectable()

export class HeroTaxReturnService {

  private currentTaxReturn!: HeroTaxReturn;

  private originalTaxReturn!: HeroTaxReturn;


  constructor(private heroService: HeroesService) { }
  set taxReturn(htr: HeroTaxReturn) {
    this.originalTaxReturn = htr;
    this.currentTaxReturn  = htr.clone();
  }

  get taxReturn(): HeroTaxReturn {
    return this.currentTaxReturn;
  }

  restoreTaxReturn() {
    this.taxReturn = this.originalTaxReturn;
  }

  saveTaxReturn() {
    this.taxReturn = this.currentTaxReturn;
    this.heroService.saveTaxReturn(this.currentTaxReturn).subscribe();
  }
}
```

Here is the HeroTaxReturnComponent that makes use of HeroTaxReturnService.

src/app/hero-tax-return.component.ts

```
content_copyimport { Component, EventEmitter, Input, Output } from
'@angular/core';
import { HeroTaxReturn } from './hero';
import { HeroTaxReturnService } from './hero-tax-return.service';

@Component({
  selector: 'app-hero-tax-return',
  templateUrl: './hero-tax-return.component.html',
  styleUrls: [ './hero-tax-return.component.css' ],
  providers: [ HeroTaxReturnService ]
})
export class HeroTaxReturnComponent {
  message = '';

  @Output() close = new EventEmitter<void>();

  get taxReturn(): HeroTaxReturn {
    return this.heroTaxReturnService.taxReturn;
  }

  @Input()
  set taxReturn(htr: HeroTaxReturn) {
```

```
  this.heroTaxReturnService.taxReturn = htr;
}

constructor(private heroTaxReturnService: HeroTaxReturnService) { }

onCanceled()  {
  this.flashMessage('Canceled');
  this.heroTaxReturnService.restoreTaxReturn();
}

onClose() { this.close.emit(); }

onSaved() {
  this.flashMessage('Saved');
  this.heroTaxReturnService.saveTaxReturn();
}

flashMessage(msg: string) {
  this.message = msg;
  setTimeout(() => this.message = '', 500);
}
```

```
}
```

The *tax-return-to-edit* arrives by way of the @Input() property, which is implemented with getters and setters. The setter initializes the component's own instance of the HeroTaxReturnService with the incoming return. The getter always returns what that service says is the current state of the hero. The component also asks the service to save and restore this tax return.

This won't work if the service is an application-wide singleton. Every component would share the same service instance, and each component would overwrite the tax return that belonged to another hero.

To prevent this, configure the component-level injector of HeroTaxReturnComponent to provide the service, using the providers property in the component metadata.

src/app/hero-tax-return.component.ts (providers)

```
content_copyproviders: [ HeroTaxReturnService ]
```

The HeroTaxReturnComponent has its own provider of the HeroTaxReturnService. Recall that every component *instance* has its own injector. Providing the service at the component level ensures that *every* instance of the component gets a private instance of the service. This makes sure that no tax return gets overwritten.

The rest of the scenario code relies on other Angular features and techniques that you can learn about elsewhere in the documentation. You can review it and download it from the live example / download example.

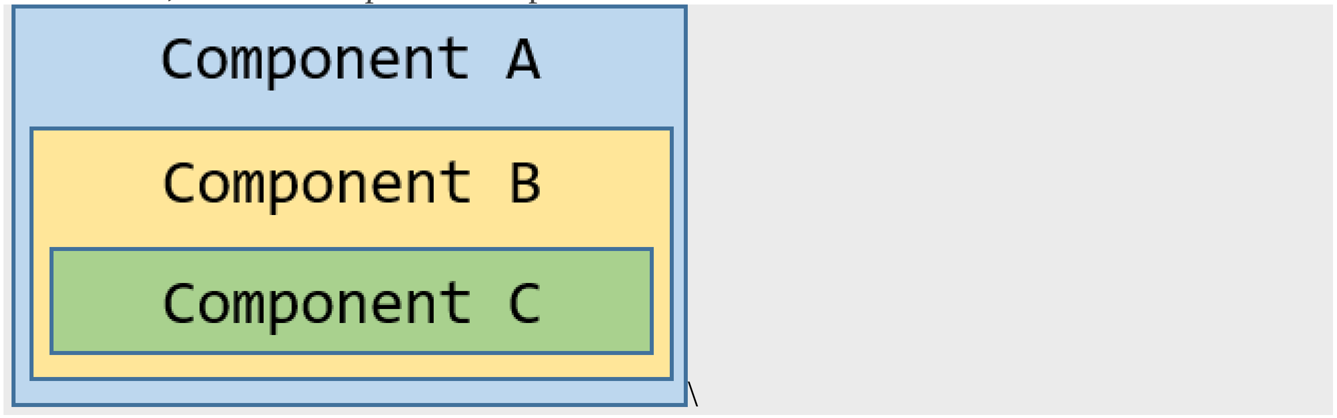
Scenario: specialized providers

Another reason to provide a service again at another level is to substitute a *more specialized* implementation of that service, deeper in the component tree.

For example, consider a Car component that includes tire service information and depends on other services to provide more details about the car.

The root injector, marked as (A), uses *generic* providers for details about CarService and EngineService.

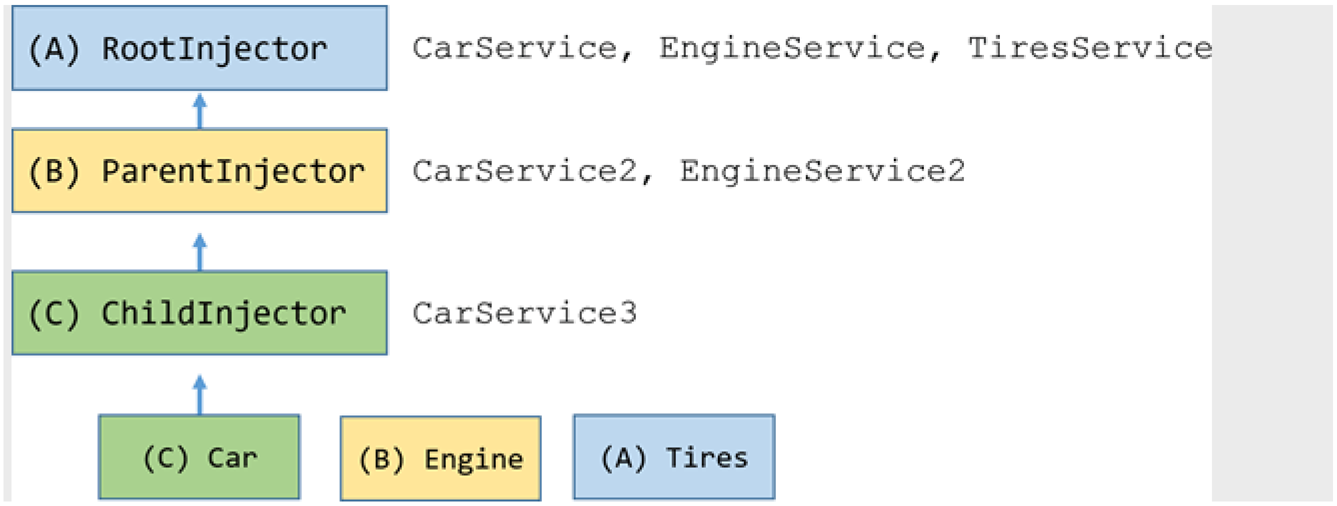
1. Car component (A). Component (A) displays tire service data about a car and specifies generic services to provide more information about the car.
2. Child component (B). Component (B) defines its own, *specialized* providers for CarService and EngineService that have special capabilities suitable for what's going on in component (B).
3. Child component (C) as a child of Component (B). Component (C) defines its own, even *more specialized* provider for CarService.



Behind the scenes, each component sets up its own injector with zero, one, or more providers defined for that component itself.

When you resolve an instance of Car at the deepest component (C), its injector produces:

- An instance of Car resolved by injector (C)
- An Engine resolved by injector (B)
- Its Tires resolved by the root injector (A).



❖ **(6)Injecting a service into another service in Angular**

say that you have a Service1 and Service2 in an regular Angular application — nothing fancy at all. Let's say now that this Service2 depends on Service1 and right way you would write something like:
And in Service2:
So far, you won't see any errors in your browser's console. But at the time that you inject Service2 into a component,
you'll see something like this:

No provider for Service1
So, whats happening here?
1 — Angular is instantiating Service2 because we injected it into AppComponent and declared it as a provider.
2 — To complete this task, Angular will check the dependencies of Service2 and, in this case, it is Service1. But, how Angular would know that Service1 is a provider to Service2? There is no providers field in @Injectable and instantiate Service1 manually it is strongly not recommended.
So what we do?
We must tell Angular to instantiate Service1 to be available (instatiated) before Service2.

Hence, we declare it as a provider into our module. In this small example application, we have only one module, the app.module.ts.
Now, Service1 is a singleton provider to the entire application and it is already instantiated. After that, Angular won't complain anymore about Service1. :)
See ya.

❖(7) Injection context:-

The dependency injection (DI) system relies internally on a runtime context where the current injector is available. This means that injectors can only work when code is executed in this context.

The injection context is available in these situations:

- Construction (via the constructor) of a class being instantiated by the DI system, such as an @Injectable or @Component.
- In the initializer for fields of such classes.
- In the factory function specified for useFactory of a Provider or an @Injectable.
- In the factory function specified for an InjectionToken.
- Within a stack frame that is run in an injection context.

Knowing when your are in an injection context, will allow you to use the inject function to inject instances.

Class constructors
Everytime the DI system instantiates a class, this is done in an injection context. This is being handled by the framework itself. The constructor of the class is executed in that runtime context thus allowing to inject a token using the inject function.

```
content_copyclass MyComponent  {

  private service1: Service1;

  private service2: Service2 = inject(Service2); // In context



  constructor() {
    this.service1 = inject(HeroService) // In context
  }
}
```

Stack frame in context
Some APIs are designed to be run in an injection context. This is the case, for example, of the router guards. It allows the use of inject to access a service within the guard function.

Here is an example for CanActivateFn

```
content_copyconst canActivateTeam: CanActivateFn =
(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) => {
return                    inject(PermissionsService).canActivate(inject(UserToken), route.params.id);
};
```

Run within an injection context
When you want to run a given function in an injection context without being in one, you can do it with runInInjectionContext. This requires to have access to a given injector like the EnvironmentInjector for example.

src/app/heroes/hero.service.ts

```
content_copy@Injectable({
  providedIn: 'root',
})
export class HeroService {
  private environmentInjector = inject(EnvironmentInjector);
```

```
    someMethod() {
      runInInjectionContext(this.environmentInjector, () => {
        inject(SomeService); // Do what you need with the injected service
      });
    }
  }
```

Note that inject will return an instance only if the injector can resolve the required token.

Asserts the context
Angular provides assertInInjectionContext helper function to assert that the current context is an injection context.

Using DI outside of a context
Calling inject or calling assertInInjectionContext outside of an injection context will throw error NG0203.

❖(8)Angular 5: Making API calls with the HttpClient service

Angular 4.3 introduced a new HttpClient service, which is a replacement for the Http service from Angular 2. It works mostly the same as the old service, handling both single and concurrent data loading with RxJs Observables, and writing data to an API.

- AngularJS
- Node.js

Metal Toad is an AWS Managed Services provider. In addition to Angular+Django work we recommend checking out our article on how to host a website on AWS in 5 minutes.

Note to readers, May 18, 2018: the code in this post is built for Angular 5.x. The same techniques will work with Angular 6 as long as you use the rxjs-compat Node package. To see how to upgrade this code for full, native RxJS compatibility, see this post.

Angular 4.3 introduced a new HttpClient service, which is a replacement for the Http service from Angular 2. It works mostly the same as the old service, handling both single and concurrent data loading with RxJs Observables, and writing data to an API.

As of Angular 5.0, the older Http service still works, but it's deprecated and has been removed in Angular 6.0. The code samples in this post are compatible with Angular 4.3 and 5.x (and 6.x with rxjs-compat). If your project is still using Angular 4.2 or lower, including Angular 2, see my previous posts on making API calls with the Http service.

About Observables and the Http service

Many JavaScript developers should be familiar with using Promises to load data asynchronously. Observables are a more feature-rich system, which emit data in packets. A single Observable object can emit a single packet of data, or can emit a stream containing multiple discrete packets. Other objects can subscribe to these Observables and run a callback each time data is emitted. (In this example using the HttpClient service, each Observable will only emit data once, but a different type of Observable could emit data more than once.)

The Observable classes in Angular are provided by the ReactiveX library.

The HttpClient service in Angular 4.3+ is the successor to Angular 2's Http service and the $http service from AngularJS 1.x. Instead of returning a Promise, its http.get() method returns an Observable object.

Try this at home!

The source code for this demo application is available on GitHub. That repository contains a simple API written in Express and a single-page Angular application which calls the API to read and write data.

This tutorial uses Webpack to manage assets. Angular 5 still supports SystemJS and you can use that instead if you prefer. For an example of how to configure SystemJS to work with HttpClient, see what it would look like if we upgraded my old Angular 2/4 demo app to Angular 5.

It's recommended that you try the Angular Tutorial first, for a basic overview of Angular architecture and Typescript.

The Back-End API
The back-end for this app is a simple Express-based API that exposes the following endpoints:

GET /api/food
Returns an array of all existing Food objects in JSON format.

GET /api/books
Returns an array of all existing Book objects in JSON format.

GET /api/movies
Returns an array of all existing Movie objects in JSON format.

POST /api/food
Creates a new Food object in the back-end data store. Accepts a JSON object in the request body.
If successful, returns a 200 OK response, containing a JSON object representing the data as saved on the server, including the auto-numbered ID

PUT /api/food/{food_id}
Updates an existing Food object. Accepts a JSON object in the request body.
If successful, returns a 200 OK response, containing a JSON object representing the data as saved on the server.

DELETE /api/food/{food_id}
Deletes an existing Food object. Does not require a response body.
If successful, returns a 200 OK response, containing a JSON object representing the Food object as it existed before it was deleted.

The entire Express API code is as follows. (Portions omitted for brevity. See the sample app for the full code.)

```
const express = require('express');
const bodyParser = require('body-parser');
const path = require('path');

const app = express();
app.use(express.static(__dirname));

app.use(bodyParser.json()); // support json encoded bodies

// some data for the API
var foods = [
  { "id": 1, "name": "Donuts" },
  { "id": 2, "name": "Pizza" },
  { "id": 3, "name": "Tacos" }
];

var books = [
  { "title": "Hitchhiker's Guide to the Galaxy" },
  { "title": "The Fellowship of the Ring" },
  { "title": "Moby Dick" }
];

var movies = [
  { "title": "Ghostbusters" },
  { "title": "Star Wars" },
  { "title": "Batman Begins" }
];

// the "index" route, which serves the Angular app
app.get('/', function (req, res) {
   res.sendFile(path.join(__dirname,'/dist/index.html'))
});

// the GET "books" API endpoint
```

```
app.get('/api/books', function (req, res) {
    res.send(books);
});

// the GET "movies" API endpoint
app.get('/api/movies', function (req, res) {
    res.send(movies);
});

// the GET "foods" API endpoint
app.get('/api/food', function (req, res) {
    res.send(foods);
});

// POST endpoint for creating a new food
app.post('/api/food', function (req, res) {
    // calculate the next ID
    let id = 1;
    if (foods.length > 0) {
        let maximum = Math.max.apply(Math, foods.map(function (f) { return f.id; }));
        id = maximum + 1;
    }
    let new_food = {"id": id, "name": req.body.name};
    foods.push(new_food);
    res.send(new_food);
});

// PUT endpoint for editing food
app.put('/api/food/:id', function (req, res) {
    let id = req.params.id;
    let f = foods.find(x => x.id == id);
    f.name = req.body.name;
    res.send(f);
});

// DELETE endpoint for deleting food
```

```
app.delete('/api/food/:id', function (req, res) {
   let id = req.params.id;
   let f = foods.find(x => x.id == id);
   foods = foods.filter(x => x.id != id);
   res.send(f);
});

// HTTP listener
app.listen(3000, function () {
   console.log('Example listening on port 3000!');
});
module.exports = app;
```

Getting Started with HttpClient

To use the Angular HttpClient, we need to inject it into our app's dependencies:

```
import { NgModule, CUSTOM_ELEMENTS_SCHEMA }     from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http';  // replaces previous Http
service
import { FormsModule } from '@angular/forms';
import { DemoService } from './demo.service';  // our custom service, see below

import { AppComponent }  from './app.component';

@NgModule({
   imports: [BrowserModule, FormsModule, HttpClientModule],
   declarations: [AppComponent],
   providers: [DemoService],
   schemas: [CUSTOM_ELEMENTS_SCHEMA],
   bootstrap: [AppComponent]
})
export class AppModule { }
```

Building the Angular Component

Our demo app contains only one simple component, which contains a few elements to display some simple data. We will be loading data from a few JSON files, to simulate an API call.

src/app/app.component.ts:

```
import {Component} from '@angular/core';
import {DemoService} from './demo.service';
import {Observable} from 'rxjs/Rx';

@Component({
  selector: 'demo-app',
  template:`
  <h1>Angular 5 HttpClient Demo App</h1>
  <h2>Foods</h2>
  <ul>
    <li *ngFor="let food of foods">{{food.name}}</li>
  </ul>
  `
})
export class AppComponent {

  public foods;

  constructor(private _demoService: DemoService) { }
}
```

Executing a Single HTTP Request

We can use HttpClient to request a single resource, by using http.get. This is very similar to the Angular 2 Http service. Notice that we no longer have to `.map((res:Response) => res.json()` because HttpClient handles this for us:

src/app/demo.service.ts:

```
import {Injectable} from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import {Observable} from 'rxjs/Observable';

const httpOptions = {
   headers: new HttpHeaders({ 'Content-Type': 'application/json' })
};
```

```
@Injectable()
export class DemoService {

    constructor(private http:HttpClient) {}

    // Uses http.get() to load data from a single API endpoint
    getFoods() {
        return this.http.get('/api/food');
    }
}
```

Our Demo service makes the HTTP request and returns the Observable object. To actually get the data from the service, we need to update our component to subscribe to the Observable:

src/app/app.component.ts:

```
...

  ngOnInit() {
    this.getFoods();
  }

+  getFoods() {
+   this._demoService.getFoods().subscribe(
+      data => { this.foods = data},
+      err => console.error(err),
+      () => console.log('done loading foods')
+    );
+  }
}
```

The subscribe() method takes three arguments which are event handlers. They are called onNext, onError, and onCompleted. The onNext method will receive the HTTP response data. Observables support streams of data and can call this event handler multiple times. In the case of the HTTP request, however, the Observable will usually emit the whole data set in one call. The onError event handler is called if the HTTP request returns an error code such as a 404. The onCompleted event handler executes after the Observable has finished returning all its data. This is less useful in the case of the Http.get() call, because all the data we need is passed into the onNext handler.

For more information about the Observable object, see the ReactiveX documentation.

In our example here, we use the onNext handler to populate the component's 'foods' variable.

The error handler just logs the error to the console. The completion callback runs after the success callback is finished.

The handler functions are optional. If you don't need the error or completion handler, you may omit them. If you don't provide an error handler, however, you may end up with an uncaught Error object which will stop execution of your application.

Executing multiple concurrent HTTP requests
Many times, we need to load data from more than one source, and we need to delay the post-loading logic until all the data has loaded. ReactiveX Observables provide a method called **forkJoin()** to wrap multiple Observables. Its subscribe() method sets the handlers on the entire set of Observables.

To run the concurrent HTTP requests, let's add the following code to our service:

src/app/demo.service.ts:

```
...

@Injectable()
export class DemoService {

  constructor(private http:HttpClient) {}

  // Uses http.get() to load data from a single API endpoint
  getFoods() {
    return this.http.get('/api/food');
  }

+   // Uses Observable.forkJoin() to run multiple concurrent http.get() requests.
+   // The entire operation will result in an error state if any single request fails.
+   getBooksAndMovies() {
+       return Observable.forkJoin(
+       this.http.get('/api/books'),
+       this.http.get('/api/movies')
+       );
+   }
}
```

Notice that forkJoin() takes multiple arguments of type Observable. These can be Http.get() calls or any other asynchronous operation which implements the Observable pattern. We don't subscribe to each of these Observables individually. Instead, we subscribe to the "container" Observable object created by forkJoin().

When using Http.get() and Observable.forkJoin() together, the onNext handler will execute only once, and only after all HTTP requests complete successfully. It will receive an array containing the combined response data from all requests. In this case, our books data will be stored in data[0] and our movies data will be stored in data[1].

The onError handler here will run if **either** of the HTTP requests returns an error code.

Next, we subscribe to the new method in our component:

src/app/app.component.ts:

```
import {Component} from '@angular/core';
import {DemoService} from './demo.service';
import {Observable} from 'rxjs/Rx';

@Component({
  selector: 'demo-app',
  template:`
  <h1>Angular 5 HttpClient Demo App</h1>
  <p>This is a complete mini-CRUD application using an Express back-end. See src/app/demo.service.ts for the API call code.</p>
  <h2>Foods</h2>
  <ul>
    <li *ngFor="let food of foods">{{food.name}}</li>
  </ul>

 <h2>Books and Movies</h2>
+    <p>This is an example of loading data from multiple endpoints using Observable.forkJoin(). The API calls here are read-only.</p>
+  <h3>Books</h3>
+  <ul>
+    <li *ngFor="let book of books">{{book.title}}</li>
+  </ul>
+  <h3>Movies</h3>
+  <ul>
+    <li *ngFor="let movie of movies">{{movie.title}}</li>
+  </ul>
  `
})
export class AppComponent {

  public foods;
+  public books;
+  public movies;

  ...

  getFoods() {
    ...
  }

+  getBooksAndMovies() {
+    this._demoService.getBooksAndMovies().subscribe(
+      data => {
+        this.books = data[0]
+        this.movies = data[1]
+      }
+      // No error or completion callbacks here. They are optional, but
+      // you will get console errors if the Observable is in an error state.
+    );
+  }

}
```

Writing data to the API

To write data to our API, we need to add several new methods to our DemoService class:

src/app/demo.service.ts:

```
@Injectable()
export class DemoService {

    constructor(private http:HttpClient) {

+    createFood(food) {
+        let body = JSON.stringify(food);
+        return this.http.post('/api/food/', body, httpOptions);
+    }
+
+    updateFood(food) {
+        let body = JSON.stringify(food);
+        return this.http.put('/api/food/' + food.id, body, httpOptions);
+    }
+
+    deleteFood(food) {
+        return this.http.delete('/api/food/' + food.id);
+    }

}
```

Notice that our createFood(), updateFood(), and deleteFood() methods use API endpoints which return the saved object in JSON format. Returning the object when creating, updating, or deleting is a nice convenience for the developer of the front-end application. Not all APIs return this data. Some may return a different status code, some XML data, or nothing at all. Consult the documentation for your API to determine what the response format will look like.

Possible bug in Firefox
At the time of this writing, the API calls on one of my projects were failing when run in Firefox. It seems that Angular 2 was not sending the Content-type: application/json headers with the requests. If your API supports this, you might be able to work around the problem by changing your API URLs to include the .json extension (e.g., /api/food/1.json).

This seems to be Firefox specific and does not affect Chrome or Edge.

Creating and Saving Data from Our Component
Now that we have the service in place, we can add some basic CRUD features to our AppComponent.

src/app/app.component.ts:

```
import {Component} from '@angular/core';
import {DemoService} from './demo.service';
import {Observable} from 'rxjs/Rx';

@Component({
  selector: 'demo-app',
  template:`
  <h1>Angular 5 HttpClient Demo App</h1>
  <p>This is a complete mini-CRUD application using an Express back-end. See src/app/demo.service.ts for the API call code.</p>
  <h2>Foods</h2>
  <ul>
-   <li *ngFor="let food of foods">{{food.name}}</li>
+         <li *ngFor="let food of foods"><input type="text" name="food-name" [(ngModel)]="food.name">
+     <button (click)="updateFood(food)">Save</button>
+     <button (click)="deleteFood(food)">Delete</button>
+   </li>
  </ul>
+       <p>Create a new food: <input type="text" name="food_name" [(ngModel)]="food_name"><button (click)="createFood(food_name)">Save</button></p>
```

```
  <h2>Books and Movies</h2>
  ...
  `
})
export class AppComponent {

  public foods;
  public books;
  public movies;

+  public food_name;

  ...

  getFoods() {
    ...
  }

  getBooksAndMovies() {
    ...
  }

+  createFood(name) {
+    let food = {name: name};
+    this._demoService.createFood(food).subscribe(
+       data => {
+         // refresh the list
+         this.getFoods();
+         return true;
+       },
+       error => {
+         console.error("Error saving food!");
+         return Observable.throw(error);
+       }
+    );
+  }
+
+  updateFood(food) {
+    this._demoService.updateFood(food).subscribe(
+       data => {
+         // refresh the list
+         this.getFoods();
+         return true;
+       },
+       error => {
+         console.error("Error saving food!");
+         return Observable.throw(error);
+       }
+    );
+  }
+
+  deleteFood(food) {
+    if (confirm("Are you sure you want to delete " + food.name + "?")) {
+      this._demoService.deleteFood(food).subscribe(
+         data => {
+           // refresh the list
+           this.getFoods();
+           return true;
+         },
+         error => {
+           console.error("Error deleting food!");
+           return Observable.throw(error);
+         }
+      );
+    }
+  }

}
```

You'll notice that we added some basic form fields and buttons to the template, and new methods createFood(), updateFood(), and deleteFood() to the component class. These are called when users click the buttons in the template, and handle saving and deleting the data.

For simplicity, I have used a simple JavaScript confirm() dialog as a delete confirmation. An enhancement might be to implement a nicer-looking dialog using another Angular component.

Can we use Observable.forkJoin() when writing to an API?
It's theoretically possible, but I wouldn't.

If you use forkJoin() to run multiple data-saving requests, it could have unpredictable results. forkJoin() will cancel all the requests if the first one returns an error. However, if one request completes successfully while a later one fails, your data could end up in a broken or partially-saved state. It seems best to use parallel API calls only when reading data.

To write multiple types of data to an API, try one of the following workflows:

1. Chain the API calls, calling each one after the previous one completes successfully, or
2. Revise your API to accept a single, larger data object, and save each piece of that larger object within the back-end code

Want More Angular?
See the next part in my series of posts about Angular API calls, showing API authentication using Django Rest Framework and JSON Web Tokens.

Looking for more examples of Angular API calls using HttpClient and ForkJoin? See how I used Angular, Django Rest Framework, HttpClient, and ForkJoin to rebuild a classic text adventure game.

See how to upgrade the code samples in this post for full, native compatibility with RxJS 6

❖(9)Create a new project

Use the ng new command to start creating your Tour of Heroes application.

This tutorial:

1. Sets up your environment.
2. Creates a new workspace and initial application project.
3. Serves the application.
4. Makes changes to the new application.

To view the application's code, see the live example / download example.

Set up your environment

To set up your development environment, follow the instructions in Local Environment Setup.

Create a new workspace and an initial application

You develop applications in the context of an Angular workspace. A *workspace* contains the files for one or more projects. A *project* is the set of files that make up an application or a library.

To create a new workspace and an initial project:

1. Ensure that you aren't already in an Angular workspace directory. For example, if you're in the Getting Started workspace from an earlier exercise, navigate to its parent.
2. Run ng new followed by the application name as shown here:
   content_copyng new angular-tour-of-heroes
3. ng new prompts you for information about features to include in the initial project. Accept the defaults by pressing the Enter or Return key.

ng new installs the necessary npm packages and other dependencies that Angular requires. This can take a few minutes.

ng new also creates the following workspace and starter project files:

- A new workspace, with a root directory named angular-tour-of-heroes
- An initial skeleton application project in the src/app subdirectory
- Related configuration files

The initial application project contains a simple application that's ready to run.

Serve the application

Go to the workspace directory and launch the application.

```
content_copycd angular-tour-of-heroes

ng serve --open
```

The ng serve command:

- Builds the application
- Starts the development server
- Watches the source files
- Rebuilds the application as you make changes

The --open flag opens a browser to http://localhost:4200.

You should see the application running in your browser.

Angular components

The page you see is the *application shell*. The shell is controlled by an Angular component named AppComponent.

Components are the fundamental building blocks of Angular applications. They display data on the screen, listen for user input, and take action based on that input.

Make changes to the application

Open the project in your favorite editor or IDE. Navigate to the src/app directory to edit the starter application. In the IDE, locate these files, which make up the AppComponent that you just created:

FILES	DETAILS
app.component.ts	The component class code, written in TypeScript.
app.component.html	The component template, written in HTML.
app.component.css	The component's private CSS styles.

When you ran ng new, Angular created test specifications for your new application. Unfortunately, making these changes breaks your newly created specifications.

That won't be a problem because Angular testing is outside the scope of this tutorial and won't be used.

To learn more about testing with Angular, see Testing.

Change the application title

Open the app.component.ts and change the title property value to 'Tour of Heroes'.

app.component.ts (class title property)

```
content_copytitle = 'Tour of Heroes';
```

Open app.component.html and delete the default template that ng new created. Replace it with the following line of HTML.

app.component.html (template)

```
content_copy<h1>{{title}}</h1>
```

The double curly braces are Angular's *interpolation binding* syntax. This interpolation binding presents the component's title property value inside the HTML header tag.

The browser refreshes and displays the new application title.

Add application styles
Most apps strive for a consistent look across the application. ng new created an empty styles.css for this purpose. Put your application-wide styles there.

Open src/styles.css and add the code below to the file.

src/styles.css (excerpt)

```
content_copy/* Application-wide Styles */

h1 {

  color: #369;

  font-family: Arial, Helvetica, sans-serif;

  font-size: 250%;

}

h2, h3 {

  color: #444;

  font-family: Arial, Helvetica, sans-serif;

  font-weight: lighter;

}

body {

  margin: 2em;

}
```

```
body, input[type="text"], button {
  color: #333;
  font-family: Cambria, Georgia, serif;
}
button {
  background-color: #eee;
  border: none;
  border-radius: 4px;
  cursor: pointer;
  color: black;
  font-size: 1.2rem;
  padding: 1rem;
  margin-right: 1rem;
  margin-bottom: 1rem;
  margin-top: 1rem;
}
button:hover {
  background-color: black;
  color: white;
}
button:disabled {
  background-color: #eee;

  color: #aaa;

  cursor: auto;

}


/* everywhere else */

* {

  font-family: Arial, Helvetica, sans-serif;

}
```

Final code review
Here are the code files discussed on this page.

src/app/app.component.ts
src/app/app.component.html
src/styles.css (excerpt)

```
content_copyimport { Component } from '@angular/core';


@Component({

  selector: 'app-root',

  templateUrl: './app.component.html',

  styleUrls: ['./app.component.css']

})

export class AppComponent {

  title = 'Tour of Heroes';
}
```