ITC003A

Building web apps with Angular

Version 14

Table of Content

Table of Content ..... **i**

Table of figures ..... **iii**

Module 1: Introduction to Angular **1**

1.1 Overview 1

1.2 Evolution in Web App development 1

1.3 Single page application (SPA) 4

1.4 Angular framework 6

Module 2: Angular core concepts **9**

2.1 Overview 9

2.2 Setup Environment 9

2.3 Angular CLI 9

2.4 Components 13

2.5 Modules 14

2.6 Component's Tree 15

2.7 Services 16

Module 3: Components and data binding **17**

3.1 Overview 17

3.2 Architecting with components 17

3.3 Data binding 19

3.4 Component communication 21

3.5 Component lifecycle 23

Module 4: Directives and elements **26**

4.1 Overview 26

4.2 Directive 26

4.3 Implementing custom directives 30

4.4 Template Variables 31

4.5 Angular elements 32

Module 5: Dependency Injection **35**

5.1 Overview 35

5.2 Dependency injection 35

5.3 Dependency injection in angular 36

Module 6: Pipes 43
6.1 Overview 43
6.2 Pipe 43
6.3 Built-In pipes 43
6.4 Usage from code 47
6.5 Custom pipes 47
6.6 pure vs impure 48
Module 7: Angular Routing 51
7.1 Overview 51
7.2 Client-side Routing 51
7.3 Router configuration 52
7.4 Common route requirements 55
7.5 Navigation 56
7.6 Read data 58
7.7 Nested navigation 59
7.8 Route guards 61
Module 8: Working with forms 64
8.1 Overview 64
8.2 Form model 64
8.3 Template-Driven 66
8.4 Model-driven 68
8.5 Data validation 70
Module 9: Reactive Programming with RxJS 75
9.1 Overview 75
9.2 Reactive programming 75
9.3 observer pattern (overview) 76
9.4 RxJS library 77
Module 10: Making HTTP Requests 89
10.1 Overview 89
10.2 HTTP Client 89
10.3 HTTP Interceptors 93

Table of figures

Figure 1 Wikipedia website 1
Figure 2 Gmail web app 2
Figure 3 jQuery example 3
Figure 4 AngularJS example 3
Figure 5 Componentizing pages 4
Figure 6 Single Page Application 5
Figure 7 Multi page application 5
Figure 8 Angular versions 7
Figure 9 Angular app building-blocks 8
Figure 10 Ng new command output 10
Figure 11 Angular project files structure 11
Figure 12 Architecting with components 17
Figure 13 Data binding 21
Figure 14 Angular component's lifecycle 23
Figure 15 ngStyle and ngClass preview 27
Figure 16 ngFor preview 29
Figure 17 ngSwitch preview 30
Figure 18 DI: Injector tree 37
Figure 19 DI: Injector scope 39
Figure 20 Pipes 43
Figure 21 Predefined date formats 46
Figure 22 Server-side routing 51
Figure 23 Client-side routing 52
Figure 24 Feature area 59
Figure 25 Multiple-level navigation 60
Figure 26 CanActivate route guard 62
Figure 27 Form validation styling 74
Figure 28 Data stream 75
Figure 29 Observer pattern 76
Figure 30 Observer pattern (UML) 77

Figure 31 Observable 78
Figure 32 RxJS Observer 79
Figure 33 RxJS operators: map 84
Figure 34 RxJS operators: filter 85
Figure 35 RxJS operators: delay 85
Figure 36 RxJS operators: merge 86
Figure 37 RxJS operators: concat 87
Figure 38 RxJS operators: catchError 87
Figure 39 HTTP Interception order 95

Module 1: Introduction to Angular

1.1 OVERVIEW

It is essential to consider why we use frameworks such as Angular, React, or Vue in the first place. Web frameworks came to rise as JavaScript became more popular and capable in the browser. This module will present a brief summary of web development which has evolved starting from websites that presented only simple information to read from users until the appearance of single page applications that are more interactive and user-focused.

After that, one of the most used frameworks in web development which is Angular will be introduced along with the presentation of its architecture and building blocks.

1.2 EVOLUTION IN WEB APP DEVELOPMENT

1.2.1 Web sites

Websites have come a long way since the beginning. The first website was a simple, static webpage with no pictures and basic font. As technologies evolve, as browsers are improved, as browsing methods change, the HTML specification grew to include all sorts of wonderful opportunities to begin laying out content. It started with table-based designs and simple multi-column web pages. Then we got CSS, which enabled us to style our content.

In the first generation of websites, users are mostly readers and are unable to contribute to the site, so there was no real interaction.

Figure 1 Wikipedia website

1.2.2 Web applications

The early days of the web saw simple web applications. A typical application had a few interactive elements in a form. The user provided information, submitted their input, and a new page loaded as a result.

Client-side scripting (JavaScript) allowed developers to create the illusion of dynamic web applications. Developers used code to show and hide elements and provide input validation before the form submit. However, until the submission of the form and the page refreshed, there could be no actual interaction.

The introduction of AJAX (Asynchronous JavaScript and Extensible Markup Language (XML)), SOAP (Simple Object Access Protocol), and REST (Representational State Transfer), broadband, and a host of other web technologies changed the way web applications work. AJAX allowed communication with servers that didn't require a full page submit and reload, so user interfaces became truly interactive.

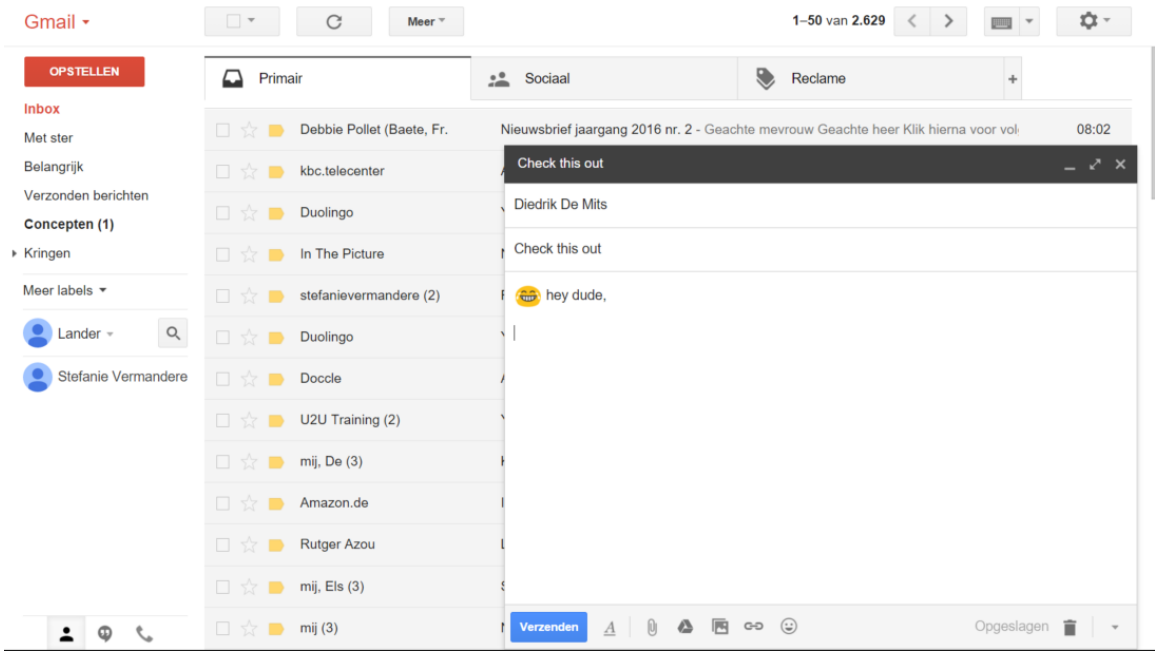


Figure 2 Gmail web app

- **First Version**

The first generation of web applications is characterized by the strong coupling between html code and JavaScript code, this type of application was developed using pure JavaScript or lightweight libraries (like jQuery) without the need for a framework. Consequently, the quality of the code is not optimal and the maintenance and testing task becomes difficult.

Code Example(jQuery):

```
function createOptions() {
  $.each(availableDates, function (key, date) {
    var value = date.toLocaleDateString();
    $('#changeDate')
       .append($("<option></option>")
       .attr("value", value)
       .text(value));
  });
};

$('#update').click(function () {
    availableDates = getDates();
    createOptions();
});
```

DOM selection

HTML generation in JS

Event Handling

Figure 3 jQuery example

- **Second version**

With the appearance of new frameworks such as Ember, AngularJS that use data binding technique, it allowed to have a better code design by separating JS code from HTML, both could evolve without having impact on each other.

Code Example (AngularJS):

```
<input type="button" ng-click="addBooking()" value="Book!" />
<table>
  <thead><tr><th>Artist</th><th>Date</th></tr></thead>
  <tbody>
    <tr ng-repeat="booking in bookings">
      <td>{{booking.artist}}</td>
      <td>{{booking.date | date:shortDate}}</td>
    </tr>
  </tbody>
</table>
```

HTML

```
app.controller('BookingsController', function ($scope) {
  $scope.current = new Booking('Jamie Cullum', $scope.availableDates[1]);

  $scope.bookings = [];

  $scope.addBooking = function () {
    $scope.bookings.push($scope.current);
    $scope.current = new Booking('', new Date());
  }
}
```

Javascript

Figure 4 AngularJS example

- **Third version**

With the latest standards of HTML and JS, new trends have emerged among the new frameworks like React, Angular and VueJS which consists of componentizing pages into smaller, independent and reusable components to reduce complexity and improve code design.

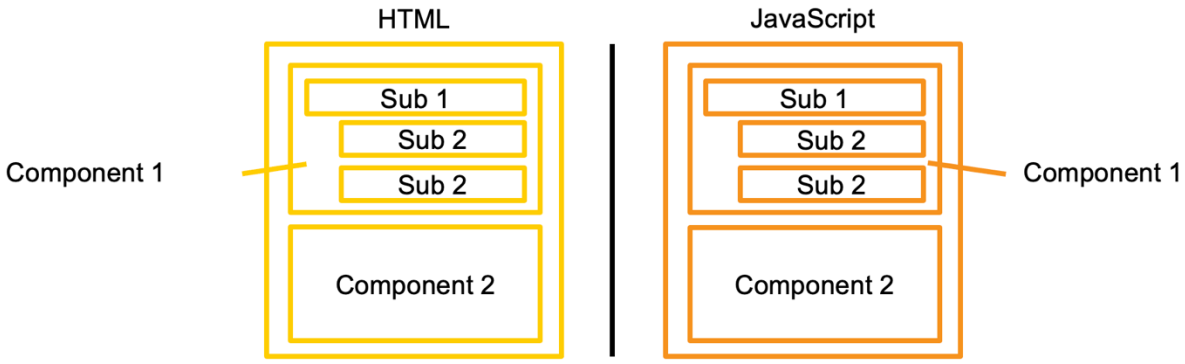


Figure 5 Componentizing pages

1.3 SINGLE PAGE APPLICATION (SPA)

1.3.1 What is a SPA?

A Single Page Application (SPA) is a single web page, website, or web application that works within a web browser and loads just a single document. It does not need page reloading during its usage, and most of its content remains the same while only some of it needs updating. When the content needs to be updated, the SPA does it through JavaScript APIs.

This way, users can view a website without loading the entire new page and data from the server. As a result, performance increases, and you feel like using a native application. It offers a more dynamic web experience to the users.

SPA requests the markup and data independently and renders pages straight in the browser. We can do this thanks to the advanced JavaScript frameworks like Angular, React, VueJS.

1.3.2 SPA vs MPA (Multi Page Applications)

The architecture of Single page applications is simple. It involves client-side technologies. Using a SPA, the server sends the HTML document only for the first request, and for subsequent requests, it sends JSON data. This means a SPA will rewrite the current page's content and not reload the whole web page. Hence, no need to wait extra for reloading and faster performance. This capability makes a SPA behave like a native application.

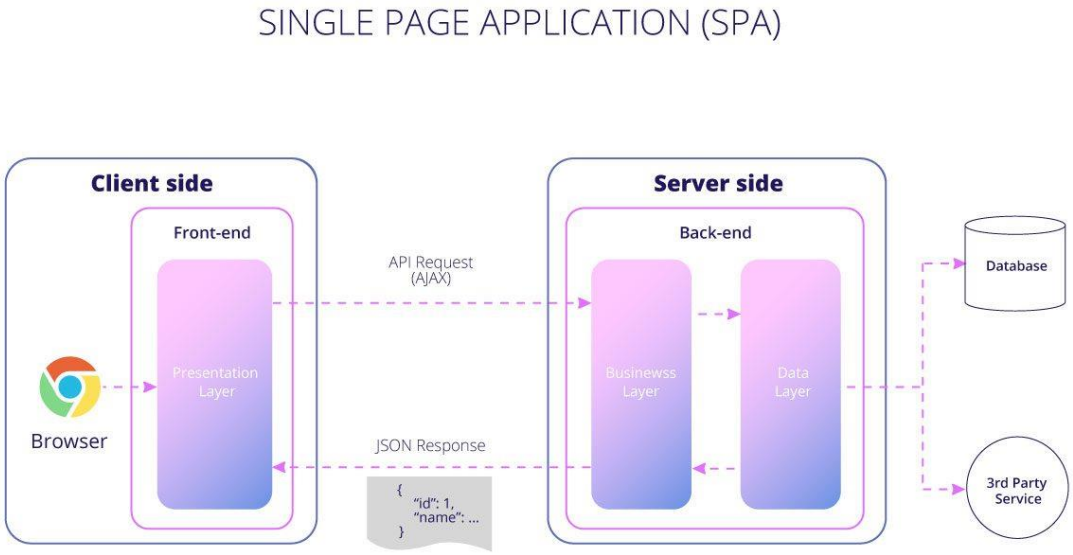


Figure 6 Single Page Application

A Single Page Application is different from multi-page applications (MPAs). The latter works in a “traditional” way. Every change e.g. display the data or submit data back to server requests rendering a new page from the server in the browser.

Thanks to AJAX, we don’t have to worry that big and complex applications have to transfer a lot of data between server and browser. That solution improves and it allows to refresh only particular parts of the application. On the other hand, it adds more complexity and it is more difficult to develop than a single-page application.

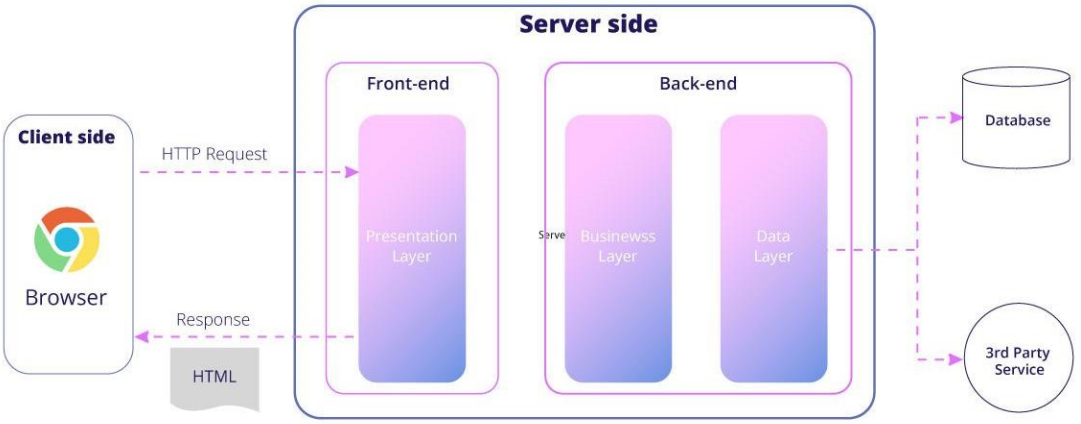


Figure 7 Multi page application

1.3.3 Benefits of using a SPA

- SPAs have App-like user experience allowing to bring more fluidity and interactivity.
- SPA is fast, as most resources (HTML, CSS, Scripts) are only loaded once throughout the lifespan of application. Only data is transmitted back and forth.
- SPAs are easy to debug with Chrome, as you can monitor network operations, investigate page elements and data associated with it.
- Offloading the server responsibility as it focuses only on building required APIs.
- It's easier to make a mobile application because the developer can reuse the same backend code for web application and native mobile application.
- SPA can cache any local storage effectively. An application sends only one request, store all data, then it can use this data and works even offline.
-

1.4 ANGULAR FRAMEWORK

1.4.1 ANGULAR

Angular is a web development platform, built on Typescript. Google maintains it, and its primary purpose is to develop single-page applications. As a framework, Angular has clear advantages while also providing a standard structure for developers to work with. It enables users to create from single-developer projects to enterprise-level applications in a maintainable manner. It includes:

- A component-based framework for building scalable web applications.
- A collection of well-integrated libraries that cover a wide variety of features, including routing, forms management, client-server communication, and more.
- A suite of developer tools to help you develop, build, test, and update your code.

1.4.2 HISTORY

"Angular" is the catch-all term for the various framework versions out there. Angular was developed in 2010, and as a result, there have been many iterations.

First, there was the original Angular, called Angular 1 and eventually known as AngularJS a JavaScript-based framework, then came Angular 2 in 2016 which is not a new version of Angular JS but a complete rewrite of the framework using Typescript.

After that, many updates are released: Angular 3, 4…, 11,12, until finally, the current version, Angular 14, released on June 2022. Each subsequent Angular version improves on its predecessor, fixing bugs, addressing issues, and accommodating increasing complexity of current platforms.

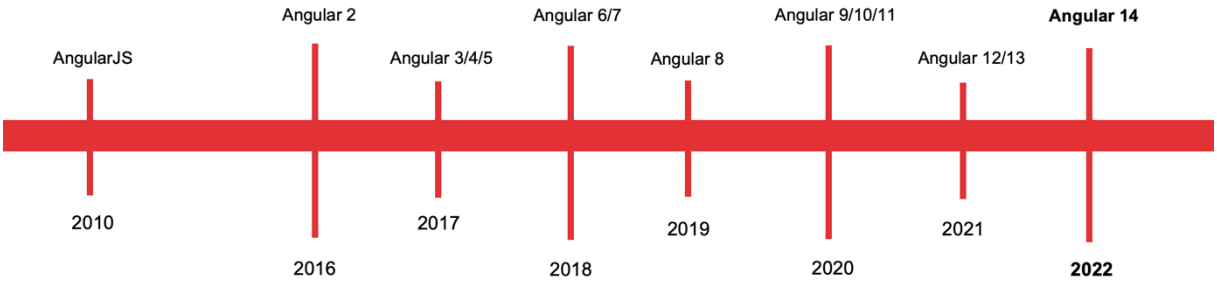


Figure 8 Angular versions

1.4.3 Angular language support

Angular differs from Angular JS in regards to programming language support. With Angular JS you generally program in JavaScript. Angular officially support three languages: JavaScript, TypeScript and Dart.

TypeScript and Dart are language super sets of JavaScript. This means they include all the features of JavaScript but add many helpful features as well. But they don't work on current browsers. To work around this limitation there are utilities to convert code written in their non-browser-compliant syntax back to JavaScript ES5. In this way they provide the developer with advanced features and still allow code to be run on existing browsers.

The process of converting code from one language to another is referred to as 'transpilation'. Technically this conversion is performed by a language pre-processor that, although it does not compile anything, is commonly referred to as a compiler.

TypeScript is an opensource language that is developed and maintained by Microsoft. It provides type safety, advanced object-oriented features, and simplified module loading. Angular itself is built using TypeScript. The documentation support is better on the Angular site for TypeScript than it is for the other options. So, although you could write your Angular apps in pure JavaScript you may find it more difficult and find less support if you get into trouble than if you spend some time learning TypeScript.

Dart is another alternative for programming Angular applications. It is similar in features to TypeScript and has its own loyal following. If your team is already versed in Dart and is looking to start developing in Angular it could be a good choice.

That being said, Typescript is the default language for Angular development.

1.4.4 Angular stylesheet support

Angular applications are styled with SASS by default. **SASS** (**S**yntactically **A**wesome **S**tyle**s**heet) which is a CSS pre-processor that provides unique features such as variables, nested rules, inline imports, and more. Angular was designed also for large scale applications, where Stylesheets could get large and complex, this is where a CSS pre-processor can help by reducing repetition of CSS and therefore saves time.

Angular also support other preprocessors like LESS or PostCSS. It's also possible to use pure CSS, that means you can apply everything you know about CSS stylesheets, selectors, rules, and media queries directly to Angular applications.

1.4.5 Angular building blocks

a) Module

Angular applications are modular and Angular has its own modularity system called ***NgModules***. NgModules are containers for a cohesive block of code dedicated to an application domain, a workflow, or a closely related set of capabilities. They can contain components, service providers.

An Angular app has a root module, named `AppModule`, which provides the bootstrap mechanism to launch the application.

b) Component

A ***component*** controls a patch of screen called a view, it consists of a TypeScript class, an HTML template, and a CSS stylesheet. The TypeScript class defines the interaction of the HTML template and the rendered DOM structure, while the style sheet describes its appearance.

c) Service

When you have data or logic that isn't associated with the view but has to be shared across components, a service class is created. It's a class that provides a number of tasks and can be reusable through the app.

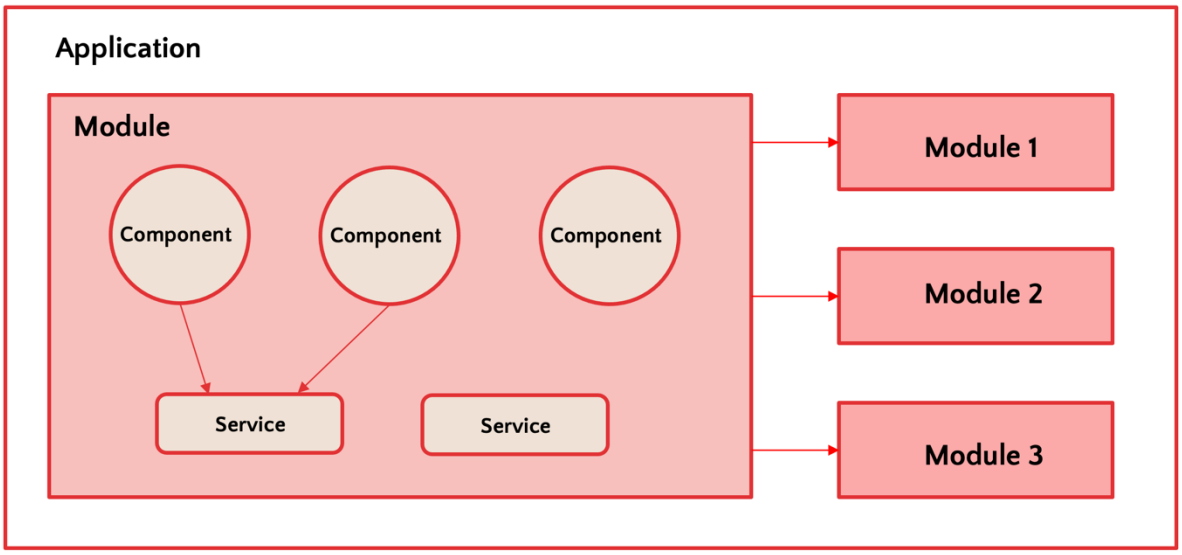


Figure 9 Angular app building-blocks

Module 2: Angular core concepts

2.1 OVERVIEW

This module introduces you to the essentials of Angular starting with setting up a local development environment, using the Angular CLI to create an Angular application and generate code. You'll have also a closer look into the fundamentals of Angular starting from components tree and modules.

2.2 SETUP ENVIRONMENT

To install Angular on your local system you need the following

Requirement	Details
Node.js	Angular requires Node.js runtime to run
NPM Package Manager	Angular applications depend on NPM dependencies for many features and functions, NPM is installed by default with Node.js

- An IDE (Integrated Development environment) like **VSCode** is also required for better development experience.

2.3 ANGULAR CLI

Angular now comes with a command line interface (CLI) to make it easier and faster to build Angular applications. **Angular CLI** is a tool that does all these things for you in some simple commands. Angular CLI uses webpack behind to do all this process.

The Angular CLI helps with:

- Bootstrapping the project
- Serving and live reloading
- Code generation
- Testing
- Packaging and releasing

2.3.1 Installing the Angular CLI

To install the CLI we use Node and **npm**:

```
npm install -g @angular/cli
```

If the above ran successfully it will have made available to you a new application called **ng**, to test this installed correctly run this command:

```
ng -v
```

2.3.2 Create a project

After installing the CLI, we can use it to create a new project using the following command:

```
ng new <project-name>
```

This outputs something like the below:

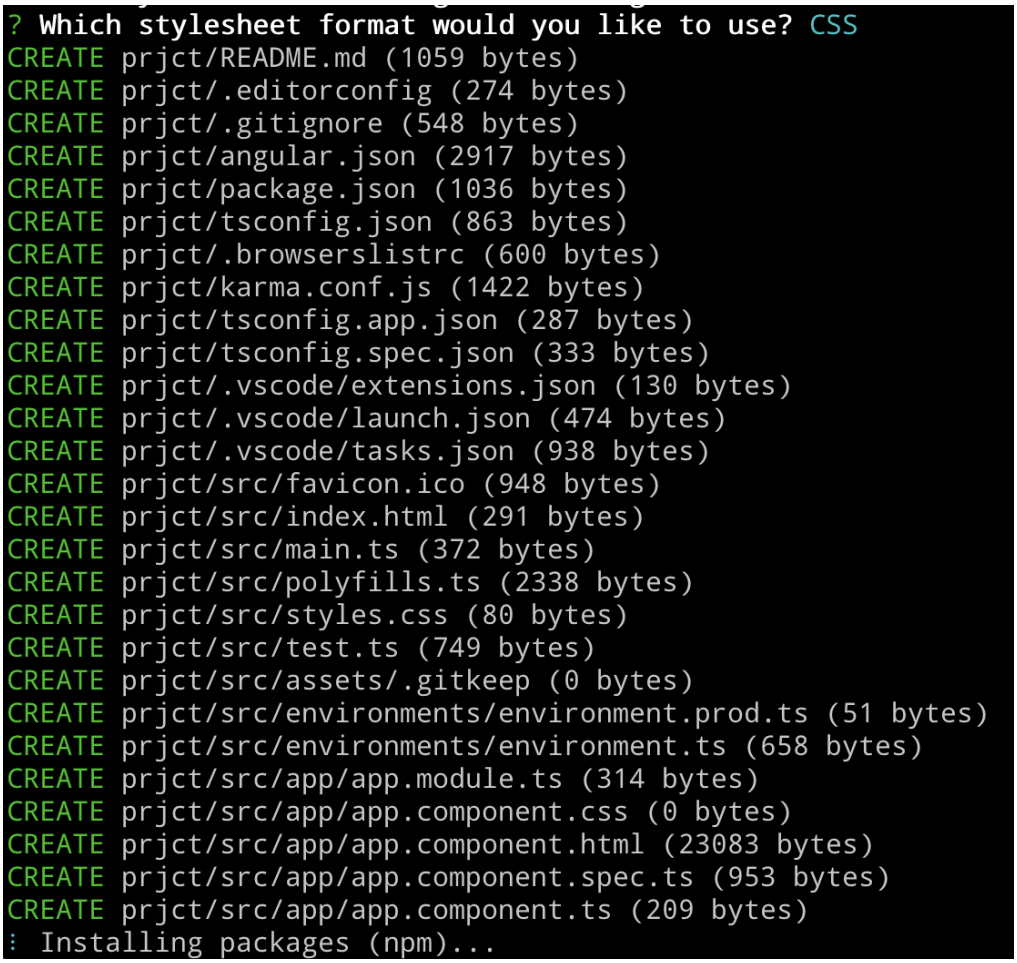


Figure 10 Ng new command output

The command generates a number of new files and folders for us:

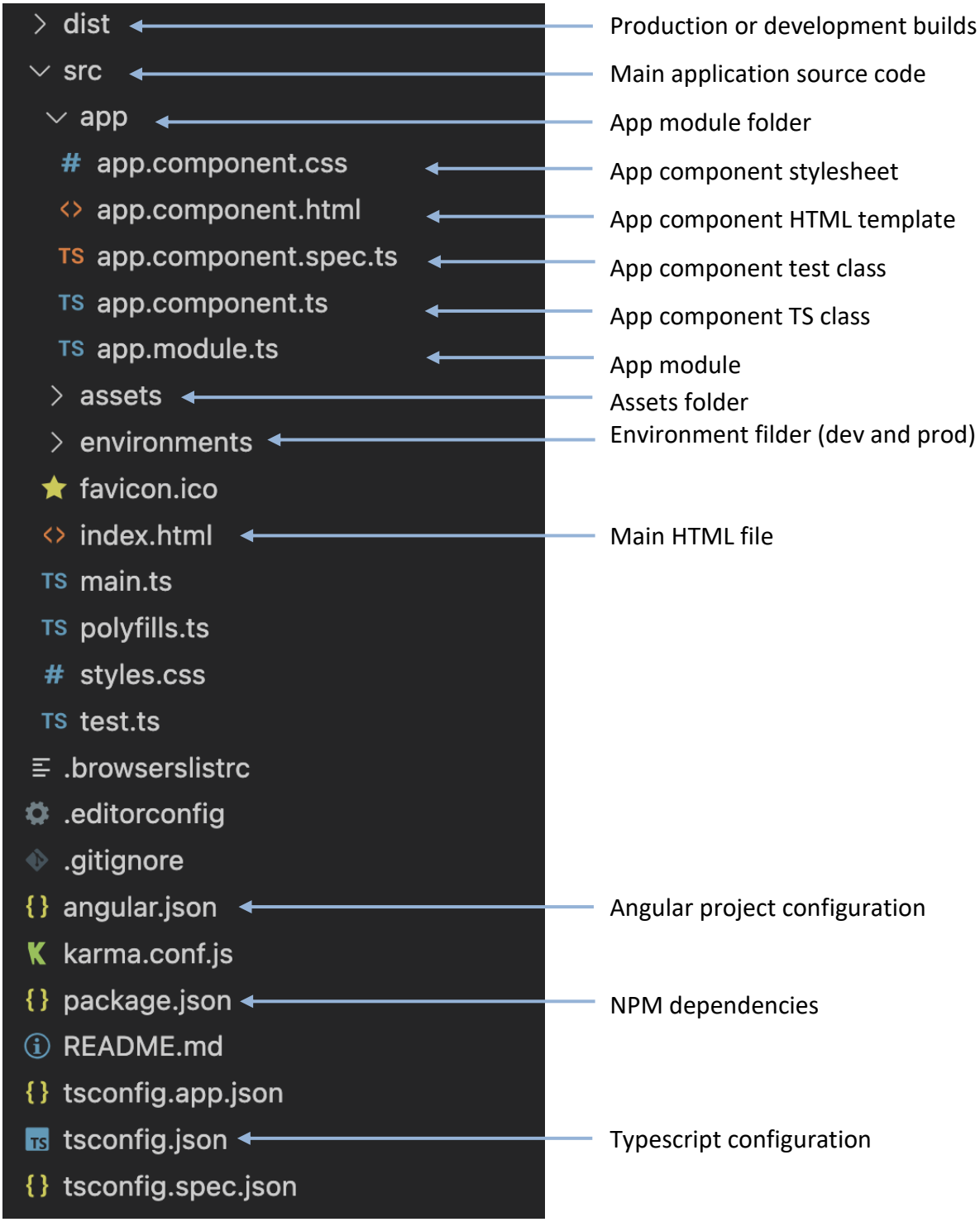


Figure 11 Angular project files structure

2.3.3 Serve a project

With the CLI we can also easily serve our application using a local web server:

```
ng serve
```

This builds our application, bundles all our code using webpack and makes it all available through ***localhost:4200***. ng serve also watches for any changes to files in our project and auto-reloads the browser for us.

We can also specify some options:

Option	Description	Default value
--open	Open the browser	
--port	Port to listen on	4200
--prod	Serve a production build	

⇒ You can find the list of all available options at the Angular official documentation

2.3.4 Build a project

To generate an application build, we run the command:

```
ng build
```

This bundles all our JavaScript, CSS, html into a smaller set of files which we can host on another site simply.

We can also specify some options:

Option	Description	Default value
--build-optimizer	Enables advanced build optimizations	
--prod	Generate a production build	
--output-path	Serve a production build	dist/

2.3.5 Generate code

The ability to generate stub code is one of the most useful features of the CLI. The most exciting part of this is that it automatically generates code that adheres to the official style guide.

Each of the below types of things it can create is called a **scaffold**. We can run this command using `ng generate <scaffold> <name>`

Available scaffolds

- **Component**

```
ng generate component <name>
```

By default, all generated files go in into `src\app\<name>`, a folder called `<name>` is created for us, it wraps the TS class file, HTML template and the stylesheet file.

- **Module**

```
ng generate module <name>
```

- **Directive**

```
ng generate directive <name>
```

- **Service**

```
ng generate service <name>
```

- **Pipe**

```
ng generate pipe <name>
```

- **Class**

```
ng generate class <name>
```

- **Interface**

```
ng generate interface <name>
```

2.4 COMPONENTS

Components are a feature of Angular that let us create a build the User interface and they are how we structure Angular applications. A component is the basic building block in Angular, it's a combination of:

- HTML Template
- Typescript Class

We use a new feature of TypeScript called annotations, and specifically an annotation (or decorator) called **`@Component`**, like so:

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

We can configure the **`@Component`** annotation by passing it an object with various parameters:

- **`selector`**: this tells Angular which tag to link this class too, for example, by setting the selector to `app-root` we've told angular that whenever it finds a tag in the HTML like `<app-root></app-root>` to use an instance of the `AppComponent` class to control it.
- **`templateUrl`**: this tells Angular where HTML template to use with the component
- **`styleUrls`**: this tells Angular what stylesheets (can be multiple) to include within the template

⇒ We can replace the parameter **`templateUrl`** by **`template`** and provide the template within the class file:

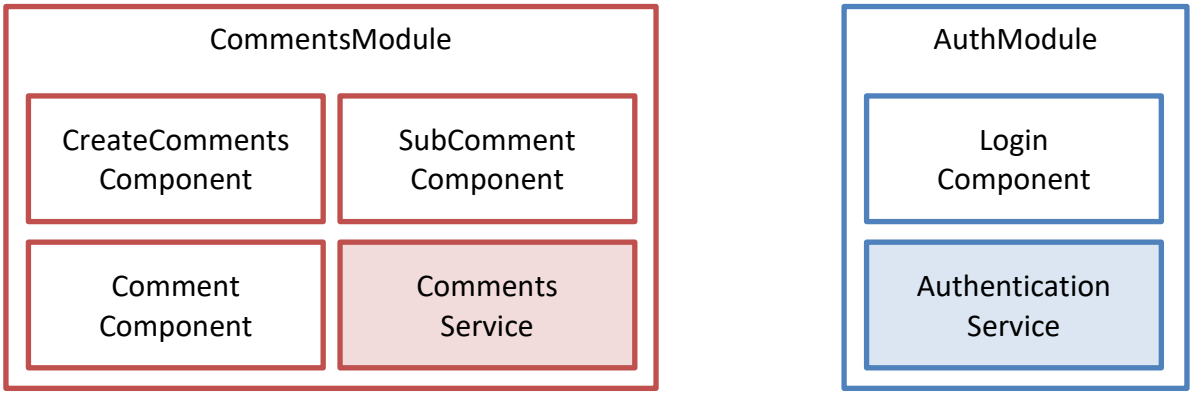
```
@Component({
    selector: 'app-root',
    template: ` <!-- Template over here -->
        <h1> This is a testing title </h1>
    `,
    styleUrls: ['./app.component.scss']
})
export class AppComponent {}
```

2.5 MODULES

In Angular your code is structured into packages called Angular Modules, or **NgModules** for short. Every app requires at least one module, the root module, that we call **`AppModule`** by convention, it is responsible for bootstrapping the app.

Modules allow to organize and structure the application, a module can contain components, services and also other nested modules.

Example:



To define an Angular Module, we first create a class and then annotate it with a decorator called **`@NgModule`**.

```
@NgModule({
    imports: [BrowserModule],
    declarations: [AppComponent],
    bootstrap: [AppComponent]
})
export class AppModule {}
```

`@NgModule` has a few params:

- **`imports`**: The other Angular Modules that export material we need in this Angular Module. Every application's root module should import the `BrowserModule`.
- **`declarations:`** the list of components, directives or pipes belonging to this module.
- **`providers`**: The set of injectable objects that are available in this module (typically Services).
- **`exports`**: exported declarations are the module's public API.
- **`bootstrap`**: Identifies the root component that Angular should bootstrap when it starts the application.

Angular itself is split into separate Angular Modules so we only need to import the ones we really use. Some other common modules you'll see in the future are the **`FormsModule`**, **`RouterModule`** and **`HttpModule`**.

2.6 COMPONENT'S TREE

An Angular application is architected as a tree of Components stemming from one root Component. Your root component is the component you configured on your root **module** in the bootstrap property, so in in the general case it's the **`AppComponent`**. By bootstrapping with **`AppComponent`** we are saying that it's the root component for our application.

In the template for our **AppComponent** we would add tags for other Components, in the template for those Components we would add tags for others... and so on and so on.

```
//app.component.html

<h1>
    This is a title
</h1>
<app-content></app-content>
<footer>
    This is a footer
</footer>
```

When bootstrapping the application, the root component (**AppComponent**) will be rendered and placed in the **index.html** file inside the **body** tag

```
//index.html

<body>
   <app-root></app-root>   //selector for AppComponent
</body>
```

⇒ **Note:** You should never modify the content of the index.html file, it should always render the root component.

2.7 SERVICES

Components shouldn't fetch or save data directly and they certainly shouldn't knowingly present fake data. They should focus on presenting data and delegate data access to a service, components deal only with the view, the other logic is delegated to services.

In Angular a service is a typescript class that can perform a number of specific tasks, and can be reusable by many components and modules.

Example:

```
@Injectable()
export class AuthenticationService {
    checkUser() {
        //TODO implements
    }
}
```

Notice that the new service imports the Angular Injectable symbol and annotates the class with the **@Injectable** decorator. This marks the class as one that participates in the dependency injection system.

Module 3: Components and data binding

3.1 OVERVIEW

Components are the main building block for Angular applications, each component consists of an HTML template and a Typescript class. In this module you will learn how to use components to architect your user interface and how to communicate between components using input data and outputs events, and finally get to know about component's lifecycle hooks that you can use to tap into key events in the lifecycle of a component initialize new instances, respond to updates during change detection, and clean up before deletion of instances.

3.2 ARCHITECTING WITH COMPONENTS

When building a new Angular application, we start by:

- Breaking down an application into separate Components.
- For each component we describe its responsibilities
- Define inputs and outputs of each component

Example:

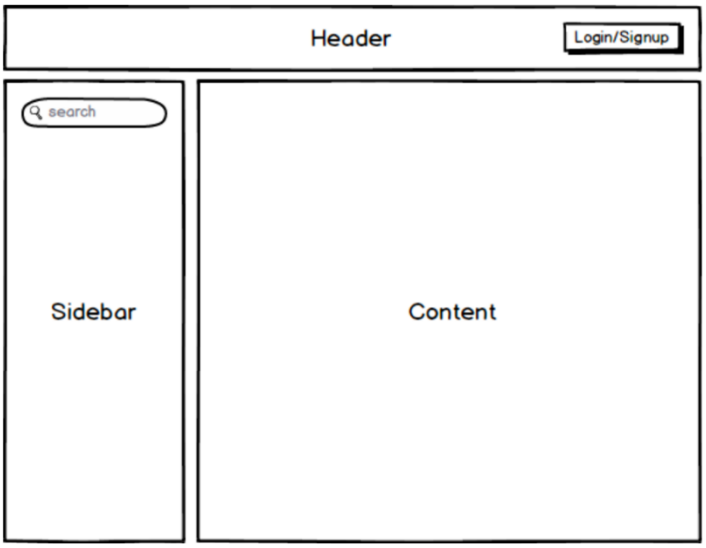


Figure 12 Architecting with components

We can split this page into three separate components:

- HeaderComponent
- SidebarComponent
- ContentComponent

AppComponent

```
<div>
    <h1>Title</h1>
    <app-header></app-header>
    <app-sidebar></app-sidebar>
    <app-content></app-content>
</div>
```

⇒ The AppComponent is the main container, it contains the Header component, Sidebar component and the Content component.

HeaderComponent

```
<div>
    <img src=”assets/logo.png”/>
    <ul>
       <li> Home </li>
       <li> Sign In </li>
       <li> … </li>
    </ul>
  </div>
```

SidebarComponent

```
<div>
    <ul>
       <li> Menu item 1 </li>
       <li> Menu item 2</li>
       <li> … </li>
    </ul>
  </div>
```

ContentComponent

```
<table *ngIf="pizzas.length > 0">
    <thead><tr><th>Name</th></tr>
    <th></th></thead>
    <tbody>
      <tr *ngFor="let pizza of pizzas">
        <td>{{pizza.name}}</td>
      </tr>
    </tbody>
  </table>
```

After creating components, they have to be declared in a module to make them available in this module, or when importing other modules, it's declarations becomes available in the current module.

Architecting an Angular application means understanding that it's just a tree of Components, each Component has some inputs and outputs and should have a clear responsibility with respect to the rest of the application.

3.3 DATA BINDING

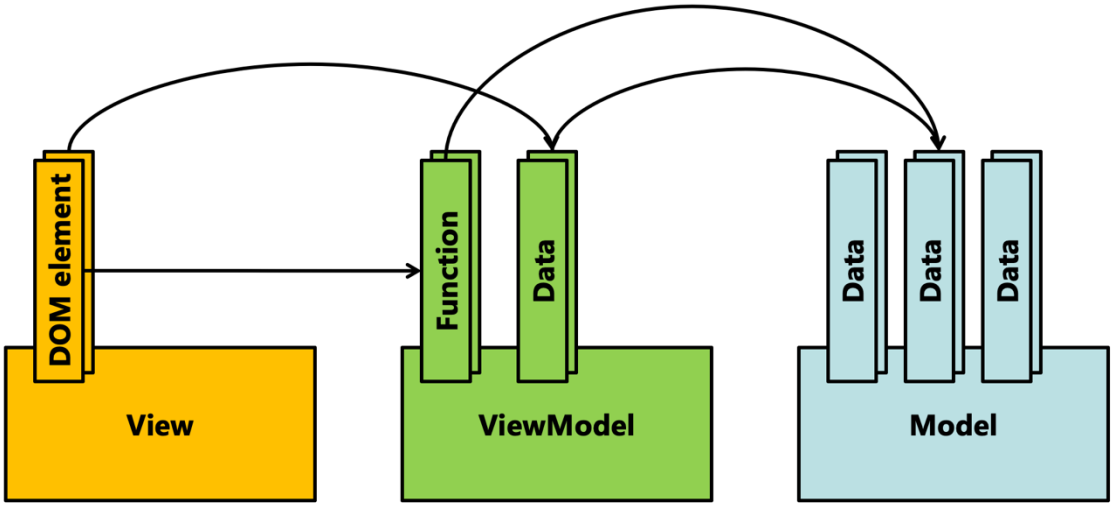
Data binding deals with how to bind your data from component to HTML DOM elements (Templates). We can easily interact with application without worrying about how to insert your data. Data binding solves the following difficulties:

- Manually setting data properties into DOM
- Synchronizing data objects with DOM
- Casting text to other types

3.3.1 MVVM Pattern

Angular framework has an MVVM software architectural setup. The MVVM model supports two-way data binding between View and ViewModel. This allows the automatic propagation to change within ViewModel's state to the view.

Data binding is the key technology that MVVM relies on, to link Views with their View-Models.



The separate code layers of MVVM are:

- **Model:** This layer is responsible for the abstraction of the data sources. Model and ViewModel work together to get and save the data.
- **View:** The purpose of this layer is to inform the ViewModel about the user's action. This layer observes the ViewModel and does not contain any kind of application logic.
- **ViewModel:** It exposes those data streams which are relevant to the View. Moreover, it serve as a link between the Model and the View.

3.3.2 One-Way data binding (Component to View)

One-way data binding is a one-way interaction between component and its template. If you perform any changes in your component, then it will reflect the HTML elements. It supports the following types:

a. String interpolation

In general, String interpolation is the process of formatting or manipulating strings. In Angular, Interpolation is used to display data from component to view (DOM). It is denoted by the expression of **{{ }}** and also known as mustache syntax.

```
<h1> {{ }} </h1>
```

b. Property binding

Property binding in Angular helps you set values for properties of HTML elements or directives. To bind to an element's property, enclose it in square brackets, **[ ]**, which identifies the property as a target property.

```
<img alt="" [src]="imageUrl"  />
```

In most cases, the target name is the name of a property, even when it appears to be the name of an attribute.

In this example, **src** is the name of the **<img>** element property. The brackets, **[ ]**, cause Angular to evaluate the right-hand side of the assignment as a dynamic expression.

c. Attribute binding

Attribute binding syntax resembles property binding, but instead of an element property between brackets, you precede the name of the attribute with the prefix **attr**, followed by a dot. Then, you set the attribute value with an expression that resolves to a string.

```
<td [attr.colspan]="2">One-Two</td></tr>
```

3.3.3 View to Component

View to components bindings are used to perform side effects, so the ViewModel could be updated, therefore certain views could be updated, there are two types:

a. Event handling Event binding lets you listen for and respond to user actions such as keystrokes, mouse movements, clicks, and touches.

To bind to an event, you use the Angular event binding syntax. This syntax consists of a target event name within parentheses to the left of an equal sign, and a quoted template statement to the right.

```
<button (click)="onSave()"> </button>
```

b. Two-way binding

Two-way binding gives components in your application a way to share data. Use two-way binding to listen for events and update values simultaneously between parent and child components.

Angular's two-way binding syntax is a combination of square brackets and parentheses, `[()]`. The `[()]` syntax combines the brackets of property binding, [], with the parentheses of event binding, (), as follows.

```
<input ([ngModel])=”hello” />
```

Summary

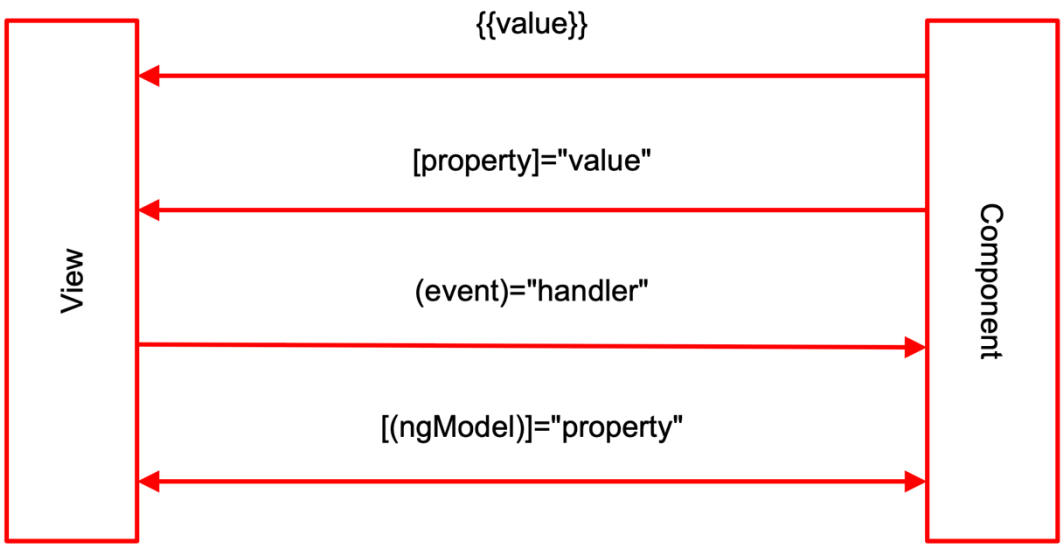


Figure 13 Data binding

3.4 COMPONENT COMMUNICATION

One common use case when we start creating components is that we want to separate the content that a component uses from the component itself.

A component is truly useful when it is reusable. One of the ways we can make a component reusable is by passing in different inputs depending on the use case. Similarly, there might be cases where we want hooks from a component when a certain activity happens within its context.

Angular provides hooks to specify each of these through decorators, aptly named Input and Output. These, unlike the **`Component`** and **`NgModule`** decorators, apply at a class member variable level

3.4.1 Inputs

Inputs are used for parent to child communication, when we add an **`@Input()`** decorator on a member variable, it automatically allows you to pass in values to the component for that particular input via Angular's property binding syntax.

TS (Child):

```
export class ArticleDetailComponent {
    @Input()  article: Article;
  }
```

HTML (Parent):

```
<article-detail [article]="current" ></article-detail>
```

3.4.2 Outputs

Just like we can pass data into a component, we can also register and listen for events from a component. We use data binding to pass data in, and we use event binding syntax to register for events. Outputs are used for child to parent communication by emitting event that will be handled by the parent, we use the **`@Output()`** decorator to accomplish this.

TS (Child):

```
export class ArticleDetailComponent {
  @Output()  onRead = new EventEmitter<Article>();
}
```

HTML (Parent):

```
<article-detail (onRead)="setAsRead()" ></article-detail>
```

When a parent needs to respond to changes of a child, we use **`EventEmitter`** class to emit the event from the child component:

```
onRead.emit(article)
```

The parent component can now respond to the event:

```
@Component({
    selector: 'my-app',
    template: `
              <div>
                <h1>Article</h1>
                <article-details (onRead)=”setAsRead($event)">
                </article-details >
              </div>
      `,
  })
  export class AppComponent {

    readArticles: Article[] = [];

    setAsRead(article: Article): void {
      this.readArticles.push(pizza);
    }
  }
```

3.5 COMPONENT LIFECYCLE

A component in Angular has a life-cycle, a number of different phases it goes through from birth to death.

We can hook into those different phases to get some pretty fine-grained control of our application. To do this we add some specific methods to our component class which get called during each of these life-cycle phases, we call those methods hooks. The hooks are executed in this order:

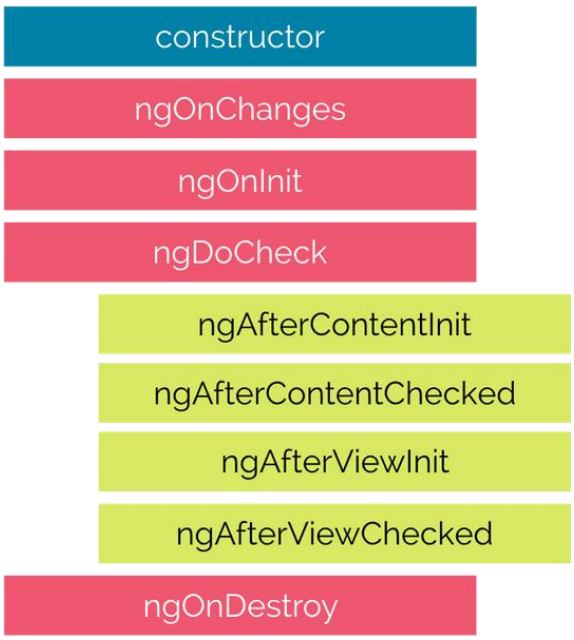


Figure 14 Angular component's lifecycle

These phases are broadly split up into phases that are linked to the component itself and phases that are linked to the children of that component.

3.5.1 Hooks for the component

Angular will first call the constructor for any component, and then the various steps mentioned earlier in order. Some of them, like the **`OnInit`** and **`AfterContentInit`** (basically, any lifecycle hook ending with Init) is called only once, when a component is initialized, while the others are called whenever any content changes. The **`OnDestroy`** hook is also called only once for a component.

Each of these lifecycle steps comes with an interface that should be implemented when a component cares about that particular lifecycle, and each interface provides a function starting with ng that needs to be implemented. For example, the **`OnInit`** life- cycle step needs a function called **`ngOnInit`** to be implemented in the component and so on.

The table below shows the interfaces and functions in the order in which they are called, along with specific details about the step if there is anything to note.

Method	Interface	Purpose
`ngOnChanges`	`OnChanges`	`ngOnChanges` is called both right after the constructor to set and then later every time the input properties to a component change. It is called before the `ngOnInit` method.
`ngOnInit`	`OnInit`	This is your typical initialization hook, allowing you to do any one-time initialization specific to your component or directive. This is the ideal place to load data from the server and so on, rather than the constructor, both for separation of concerns as well as testability.
`ngDoCheck`	`DoCheck`	`DoCheck` is Angular's way of giving the component a way to check if there are any bindings or changes that Angular can't or should not detect on its own. This is one of the ways we can use to notify Angular of a change in the component.
`ngAfterContentInit`	`AfterContentInit`	As mentioned, the `AfterContentInit` hook is triggered during component projection cases, and only once during initialization of the component. If there is no projection, this is triggered immediately.

ngAfterContentChecked	AfterContentChecked	AfterContentChecked is triggered each time Angular's change detection cycle executes, and in case it is initialization, it is triggered right after the AfterContentInit hook.
ngAfterViewInit	AfterViewInit	AfterViewInit is the complement to AfterContentInit, and is triggered after all the child components that are directly used in the template of the component are finished initializing and their views updated with bindings. This may not necessarily mean that the views are rendered into the browser, but that Angular has finished updating its internal views to render as soon as possible. AfterViewInit is triggered only once during the load of the component.
ngAfterViewChecked	AfterViewChecked	AfterViewChecked is triggered each time after all the child components have been checked and updated. Again, a good way to think about both this and AfterContentChecked is like a depth-first tree traversal, in that it will execute only after all the children components' AfterViewChecked hooks have finished executing.
ngOnDestroy	OnDestroy	The OnDestroy hook is called when a component is about to be destroyed and removed from the UI. It is a good place to do all cleanup, like unsubscribing any listeners you may have initialized and the like. It is generally good practice to clean up anything that you have registered (timers, observables, etc.) as part of the component.

Module 4: Directives and elements

4.1 OVERVIEW

In Angular, Directives are defined as classes that can add new behavior to the elements in the template or modify existing behavior. In this module you'll learn how to use built-in Angular directives and make the difference between the two types of directives: Attribute and Structural. After that new Angular elements will be introduced, which are commonly used besides native HTML elements.

4.2 DIRECTIVE

A directive allows you to attach some custom functionality to elements in your HTML. A component in Angular, like the one we built in the previous chapter, is a directive that provides both functionality and UI logic. It is fundamentally an element that encapsulates its behavior and rendering logic. These can be further classified into two types:

4.2.1 ATTRIBUTE DIRECTIVES

Attribute directives change the look and feel, or the behavior, of an existing element or component that it is applied on. There are two basic attribute directives that Angular provides out of the box, which are the **`ngClass`** and the **`ngStyle`** directives. Both of these are alternatives for the class and style bindings:

a. `ngClass`

The `ngClass` directive allows us to apply or remove multiple CSS classes simultaneously from an element in our HTML. Angular provides the `ngClass` directive, which can take a JavaScript object as input.

```
[ngClass]="{'text-success': true}"
```

For each key in the object that has a truthy value, Angular will add that key (the key itself, not the value of the key) as a class to the element. Similarly, each key in the object that has a falsy value will be removed as a class from that element.

Since the object literal can contain many keys, we can also set many class names.

```
<h4>NgClass</h4>
<ul *ngFor="let person of people">
  <li [ngClass]="{
      'text-success':person.country === 'UK',
      'text-primary':person.country === 'USA',
      'text-danger':person.country === 'HK'
    }">{{ person.name }} ({{ person.country }})
  </li>
</ul>
```

b. `ngStyle`

The `ngStyle` directive is the lower-level equivalent of the `ngClass` directive. It operates in a manner similar to the `ngClass` in that it takes a JSON object and applies it based on the values of the keys. But the `ngStyle` directive works at a CSS style/properties level. The keys and values it expects are CSS properties and attributes rather than class names.

One way to set styles is by using the `ngStyle` directive and assigning it an object literal, like so:

```
<div [ngStyle]="{'background-color':'green'}"></div>
```

This sets the background color of the div to green.

`ngStyle` becomes much more useful when the value is dynamic. The values in the object literal that we assign to `ngStyle` can be JavaScript expressions which are evaluated and the result of that expression is used as the value of the CSS property, like this:

```
<div [ngStyle]="{'background-color’: person. country == 'UK’? 'green': 'red'}">
</div>
```

The above code uses the ternary operator to set the background color to green if the persons country is the UK else red.

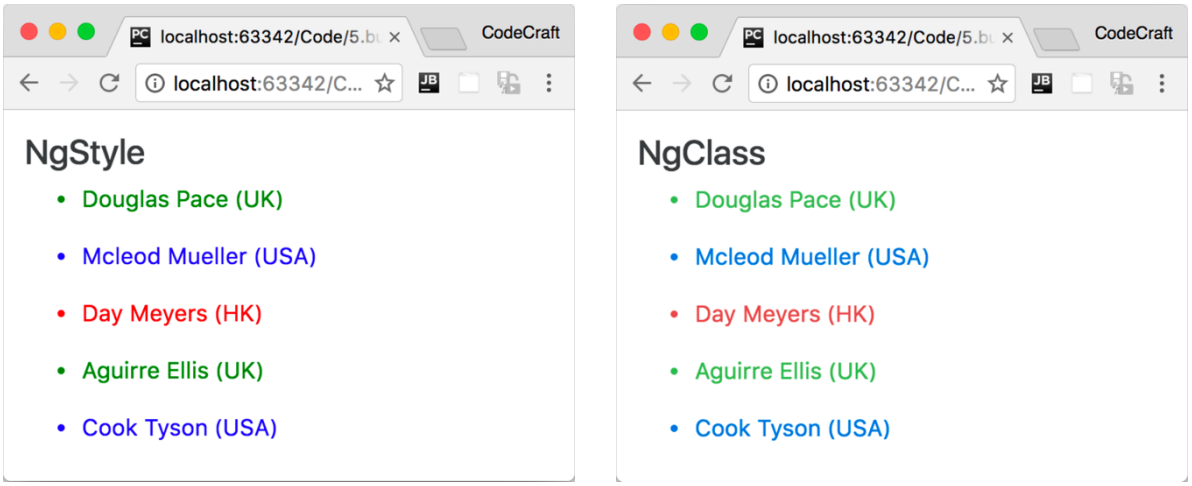


Figure 15 ngStyle and ngClass preview

4.2.2 STRUCTURAL DIRECTIVES

Structural directives change the DOM layout by adding or removing elements from the view. **`ngIf`** and **`ngFor`** are examples of structural directives.

Structural directives, as discussed earlier, are responsible for changing the layout of the HTML by adding, removing, or modifying elements from the DOM. Just like other directives that are not components, structural directives are applied on a pre-existing element, and the directive then operates on the content of that element.

Structural directives in Angular follow a very particular syntax, which makes it easy to recognize when a directive is a structural directive versus an attribute one. All structural directives in Angular start with an asterisk (`*`).

a. `*ngIf`

We will first take a look at the trusty and often used structural directive `ngIf`. The `ngIf` directive allows you to conditionally hide or show elements in your UI. The syntax, as mentioned earlier, starts with an asterisk as it is a structural directive that can conditionally remove or add elements to our rendered HTML.

The `ngIf` uses one of the simplest syntaxes of all the structural directives, as it simply expects the expression provided to it to evaluate to a truthy value. This is the JavaScript concept of truthiness (as explained previously), so the Boolean true value, a nonzero number, a nonempty string, and a nonnull object would all be treated as true. This also makes it convenient to have templates that show up if certain objects are present and nonnull.

```
<div *ngIf="person.age>30"></div>
```

If the condition is false the element the directive is attached to will be removed from the DOM.

b. `*ngFor`

While the `ngIf` directive is used for conditionally showing/hiding elements, the `ngFor` directive is used for creating multiple elements, usually one for each instance of some or the other object in an array. It is a common practice to have a template, and then create an instance of that template for each instance of our object.

The syntax is `*ngFor="let <value> of <collection>"`.

`<value>` is a variable name of your choosing, *`<collection>`* is a property on your component which holds a collection, usually an array but anything that can be iterated over in a for-of loop.

```
<ul>
  <li *ngFor="let person of people">
    {{ person.name }}
  </li>
</ul>
```

If we ran the above, we would see this:

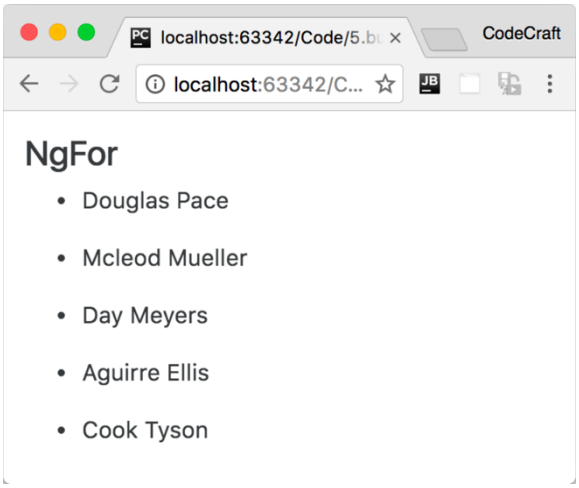


Figure 16 ngFor preview

c. ngSwitch

The last built-in directive is actually a set of multiple directives. `ngSwitch` by itself is not a structural directive, but rather an attribute directive. You would use it with normal data-binding syntax with the square-bracket notation. It is the `ngSwitchCase` and the `ngSwitchDefault` directives that are, in fact, structural directives, as they would add or remove elements depending on the case.

Let's imagine we wanted to print people names in different colors depending on where they are. Green for UK, Blue for USA, Red for HK. With bootstrap we can change the text color by using the `text-danger`, `text-success`, `text-warning` and `text-primary` classes.

```
<ul *ngFor="let person of people" [ngSwitch]="person.country">
  <li *ngSwitchCase="'UK'"  class="text-success">
      {{ person.name }} ({{ person.country }})
  </li>
  <li *ngSwitchCase="'USA'" class="text-primary">
      {{ person.name }} ({{ person.country }})
  </li>
  <li *ngSwitchCase="'HK'" class="text-danger">
      {{ person.name }} ({{ person.country }})
  </li>
```

- We bind an expression to the `ngSwitch` directive.
- The `ngSwitchCase` directive lets us define a condition which if it matches the expression will render the element it's attached to.
- If no conditions are met in the switch statement it will check to see if there is an `ngSwitchDefault` directive, if there is it will render the element that's attached to, however it is optional - if it's not present it simply won't display anything if no matching `ngSwitchCase` directive is found.

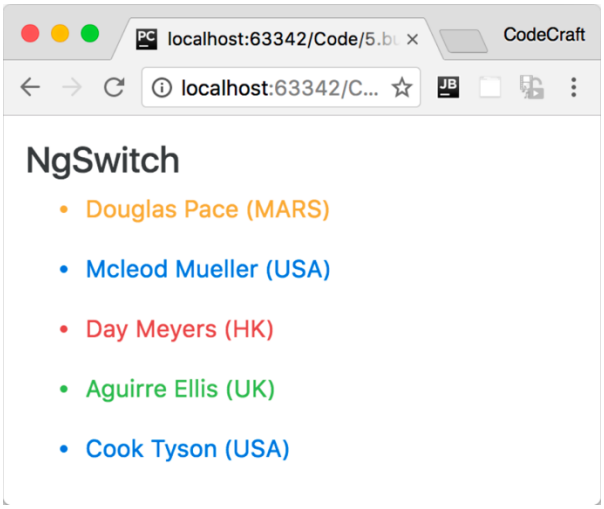


Figure 17 ngSwitch preview

⇒ The use of `ngSwitch` here is just for example and isn't an efficient way of solving this problem. We would use either the `ngStyle` or `ngClass` directives.

4.3 IMPLEMENTING CUSTOM DIRECTIVES

Creating a custom directive is just like creating an Angular component. To create a custom directive, we have to use the **`@Directive`** decorator.

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

@Directive({
  selector:"[myHighlight]"
})
export class MyHighlightDirective { }
```

The `@Directive` decorator has a `selector` option just like the `@Component` decorator, for attribute directive the `selector` must be between bracket `[<name>]` to indicate that each element that has the attribute <name> this directive should be applied to it.

When creating a directive, it must be part of a module, this is how it knows which class to invoke whenever it encounters the selector in the template.

```
@NgModule({
    declarations: [
      AppComponent,
      MyHighlightDirective
    ],
  })
  export class AppModule { }
```

Once we have our decorator added, the next step would be to add a constructor to our directive, as shown below:

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

@Directive({
  selector:"[myHighlight]"
})
export class MyHighlightDirective {

    constructor(private elRef: ElementRef) {}
 }
```

With the above code, we are telling Angular to inject an instance of `ElementRef` into its constructor, whenever the directive is created. This is actually via dependency injection.

`ElementRef` is used to get direct access to the DOM element on which our directive attribute is attached to.

Let's say we now want to change the background color of this element to *green* and the foreground color to *blue*. To do that, we'll write the following code:

```
@Directive({
    selector:"[myHighlight]"
  })

  export class MyHighlightDirective {

      constructor(private elRef: ElementRef) {}

      ngOnInit() {
          this.elRef.nativeElement.style.backgroundColor = "green";
          this.elRef.nativeElement.style.color = "blue";
        }

  }
```

Now, we're able to use our directive:

```
<div class="float-left" myHighlight  >Some text to be highlighted !</div>
```

4.4 TEMPLATE VARIABLES

Template variables help you use data from one part of a template in another part of the template. Use template variables to perform tasks such as respond to user input or finely tune your application's forms.

A template variable can refer to the following:

- A DOM element within a template
- A directive or component
- A `TemplateRef` from an `ng-template` (See next section)

In the template, you use the hash symbol, `#`, to declare a template variable. The following template variable, `#phone`, declares a phone variable with the `<input>` element as its value.

```
<input #phone placeholder="phone number" />
```

Angular assigns a template variable a value based on where you declare the variable:

- If you declare the variable on a component, the variable refers to the component instance.
- If you declare the variable on a standard HTML tag, the variable refers to the element.
- If you declare the variable on an `<ng-template>` element, the variable refers to a `TemplateRef` instance which represents the template. (See next section)

4.5 ANGULAR ELEMENTS

4.5.1 `ng-container`

ng-container is an extremely simple directive that allows you to group elements in a template that doesn't interfere with styles or layout because Angular doesn't put it in the DOM

This component is used when you do not want to add an extra HTML element to the DOM (like a div or span), but you want a wrapper around children components.

`ng-container` work just like that, and it also accepts Angular directives (`ngIf`, `ngFor`, etc.). They are elements that can serve as wrappers but do not add an extra element to the DOM.

Let's see an example where you might want to use this:

```
<ul *ngIf="store.products" *ngFor="let product of store.products">
  <li>{{ product.name }}</li>
</ul>
```

As you may already know, in Angular, you cannot apply two or more structural directives on an element. Your next solution may be:

```
<div *ngIf="store.products">
    <ul *ngFor="let product of store.products">
      <li>{{ product.name }}</li>
    </ul>
  </div>
```

This code would work, but now you're introducing a div element that you may not need. We can solve this with ng-container like so:

```
<ng-container *ngIf="store.products">
    <ul *ngFor="let product of store.products">
      <li>{{ product.name }}</li>
    </ul>
  </ng-container>
```

With this, the DOM won't include the `ng-container`, so we're not adding an extra element that we do not need. Angular will render the container element as a comment in the DOM. And if you need to use another structural directive before the `ul` element, you can use more `ng-container`.

4.5.2 ng-template

As the name suggests the `ng-template` is a template element that Angular uses with structural directives (`ngIf`, `ngFor`, `ngSwitch` and custom directives). Angular does not render it by default. These template elements only work in the presence of structural directives, which help us to define a template that doesn't render anything by itself, but conditionally renders them to the DOM. It helps us create dynamic templates that can be customized and configured.

With the following code:

```
<h1>Hello</h1>

<ng-template>
  <h2>How are you</h2>
</ng-template>
```

If you run the previous code and inspect the HMTL source code in the browser, you'll find the second part commented like this:

```
<h1>Hello</h1>

<!-- -->
```

It is not rendered; it's only defined in the source code.

Let's look at a use case for this element:

```
<ng-template #noProducts>
    <p>There are no products in this store</p>
  </ng-template>
```

Above, we've defined a template with the name *noProducts* using the Hashtag '#'. In this example, we'll render it to the DOM if there are no products, like this:

```
<ng-container *ngIf="store.products">
    <ng-container *ngIf="store.products.length > 0 else noProducts">
      <ul *ngFor="let product of store.products">
        <li>{{ product.name }}</li>
      </ul>
    </ng-container>
  </ng-container>

  <ng-template #noProducts>
    <p>There are no products in this store</p>
  </ng-template>
```

4.5.3 ng-content

ng-content is used to project content into Angular components. You use the <ng-content></ng-content> tag as a placeholder for that dynamic content, then when the template is parsed Angular will replace that placeholder tag with your content.

For example, you have two components as parent and child component and want to show some data in the child component from the parent component.

In *parent.component.html* *<app-child>* selector is used to show data of child component

```
<app-child>
    <div> Child Component Details </div>
  </app-child>
```

If you check on your browser ** Child Component Details ** inside <app-child></app-child> would not be visible. What if we want to show this content? So this is where the ng-content directive comes into the picture.

What we need to do is, just add "ng-content" inside the component template and it will find the content inside the directive tag and add it to that template at that particular place where we added the "ng-content" tag.

So instead of div, you do something similar with Angular components except if you tell Angular where to display it in the parent template using ng-content.

In child.component.html:

```
<h1>Child Info</h1>
<ng-content></ng-content>
```

When rendering Angular will replace <ng-content></ng-content> tag with the actual content passed by the parent component, between the opening and the closing tag of the child component.

Module 5: Dependency Injection

5.1 OVERVIEW

Service is a broad category encompassing any value, function, or feature that an application need. A service is typically a class with a narrow, well-defined purpose. It should do something specific and do it well. Angular distinguishes components from services to increase modularity and reusability. A component should use services for tasks that don't involve the view or application logic. Services are good for tasks such as fetching data from the server, validating user input, or logging directly to the console.

In Angular, dependency injection makes those services available to components, Angular uses the Dependency Injection design pattern, which makes it extremely efficient. This programming paradigm allows classes, components, and modules to be interdependent while maintaining consistency.

This module will cover a brief definition of what dependency injection is and how Angular facilitates the interaction between dependency consumers (Components) and dependency providers using an abstraction called Injector.

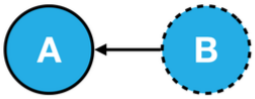
5.2 DEPENDENCY INJECTION

Before we dive deeper into services and other related topics, let’s take a step back to understand dependency injection, especially in context to Angular.

Dependency injection started in static languages that are more common in server-side programming. In simple terms, dependency injection is the idea that any class or function should ask for its dependencies, rather than instantiating it themselves. Something else (usually called an injector) would be responsible for figuring out what is needed and how to instantiate it.

What is dependency?

When module A in an application needs module B to run, then module B is a dependency of module A.



Dependency injection has huge benefits when we practice it in our applications, as it allows us to create modular, reusable pieces while allowing us to test components and modules easily. Let’s take a simple example to demonstrate how dependency injection can make your code more modular, easier to change, and testable:

```
class MyDummyService {
    getMyData() {
        let httpService = new HttpService();
        return httpService.get('my/api');
    }
}

class MyDIService {
    constructor(private httpService: HttpService) { }

    getMyData() {
        return this.httpService.get('my/api');
    }
}
```

In this example, there is actually not much to differentiate between `MyDummyService` and `MyDIService`, except for the fact that one instantiates an `HttpService` before using it, while the other asks for an instance of it in the constructor. But this small change allows for many things, including:

- It makes it more obvious what is necessary for each service to actually execute, rather than finding out at the time of execution.
- In our example, instantiating the `HttpService` was trivial, but it might not be in certain cases. In such a case, every user of `HttpService` will need to know exactly how to create it and configure it, before using it.
- In our test, we might not want to make actual HTTP calls. There, we can replace and instantiate `MyDIService` with a fake HttpService that does not make real calls, while there is nothing, we can do for `MyDummyService`.

5.3 DEPENDENCY INJECTION IN ANGULAR

Angular has set up its dependency injection system, and the major things developers should be aware of. We will not go into each and every detail in this section, but cover the general aspects we expect most developers to encounter in their day-to-day work.

The DI framework in Angular consists of 3 concepts working together:

5.3.1 INJECTOR

The injector is the mechanism that creates an object of the service class, and injects that object to a client object. In this way, the DI pattern separates the responsibility of creating an object of the service class out of the client class.

The injector is the main mechanism. Angular creates an application-wide injector for you during the bootstrap process, and additional injectors as needed. You don't have to create injectors.

The injector tree

An Angular application will have a tree of injectors mirroring the component tree. We have a top-level parent injector which is attached to our `NgModule`. Then we have child injectors descending in a hierarchy matching the component tree.

So, a parent component will have a child injector stemming from `NgModule` and A child component of parent component will have a child injector stemming from Parent.

An injector creates dependencies and maintains a container of dependency instances that it reuses, if possible.

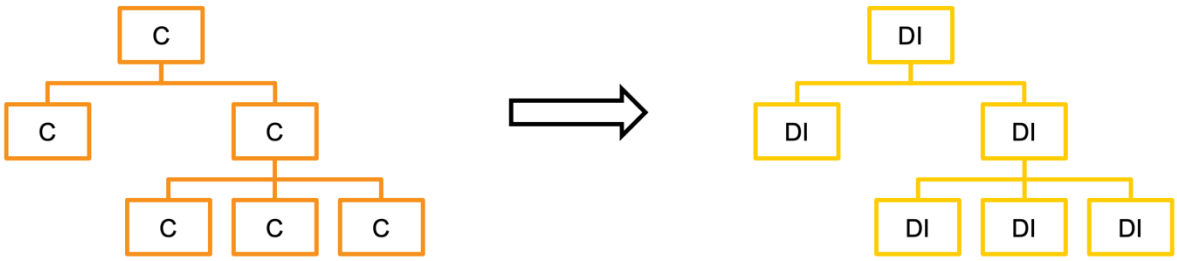


Figure 18 DI: Injector tree

5.3.2 DEPENDENCIES (SERVICES)

We have mostly worked with Angular components so far. Components, to quickly recap, are responsible for deciding what data to display and how to render and display it in the UI. We bind our data from the components to our UI and bind our events from the UI to methods in the components to allow and handle user interactions. That is, components in Angular are our presentation layer, and should be involved in and focus on the presentation aspects of data.

But if components are our presentation layer, it begs the question of what should be responsible for the actual data fetching and common business logic in an Angular application. This is where Angular services come in. Angular services are that layer that is common across your application, that can be reused across various components. Generally, you would create and use Angular services when:

- You need to retrieve data from or send data to your server. This may or may not involve any processing of the data while it is being transferred.
- You need to encapsulate application logic that is not specific to any one component, or logic that can be reused across components.
- You need to share data across components, especially across components that may or may not know about each other. Services by default are singletons across your application, which allows you to store state and access them across various components.

Another simple way to think about Angular services is that it is the layer to abstract the “how” away from the component, so that the component can just focus on the “what,” and let the service decide the how.

We can use Angular CLI to generate a service like the following:

```
ng generate service –module=app
```

`@Injectable` decorator

A service has to be annotated using `@Injectable` decorator, it defines a class as a service in Angular and allows Angular to inject it into a component as a dependency. Likewise, the `@Injectable` decorator indicates that a component, class, pipe, or `@NgModule` has a dependency on a service.

For any dependency that you need in your app, you must register a provider with the application's injector, so that the injector can use the provider to create new instances. For a service, the provider is typically the service class itself.

Providing services

- By default, the Angular CLI command ng generate service registers a provider with the root injector for your service by including provider metadata in the `@Injectable` decorator.

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

- When you register a provider with a specific `NgModule` , the same instance of a service is available to all components in that NgModule. To register at this level, use the `providers` property of the `@NgModule` decorator.

```
@NgModule({
    providers: [
       BackendService,
       Logger
   ],
```

- 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 component level, register a service provider in the providers property of the `@Component` metadata.

```
@Component({
    selector:    'app-hero-list',
    templateUrl: './hero-list.component.html',
    providers:  [ HeroService ]
  })
```

- When registered in a component the service is a singleton in the **scope If the injector,** so it can be injected into the components and all its children

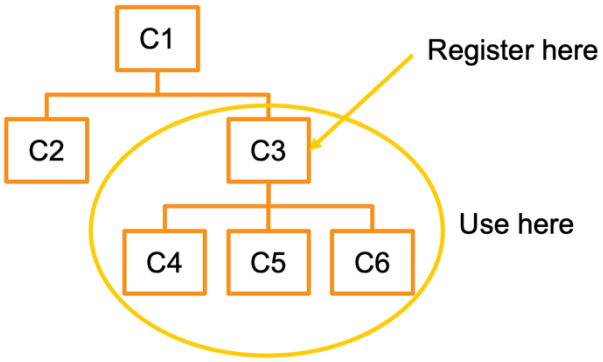


Figure 19 DI: Injector scope

Using services

To use a service, you have to inject it into the constructor of the component (or other service) in-need:

```
constructor(private service: LoggerService) { }
```

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 makes 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.

Optional dependency

When a dependency (Service) not found, meaning that none of the injectors could resolve, Angular throws an Error by default.

```
EXCEPTION: No provider for LoggerService! (GameListComponent ->
GameService -> LoggerService)
```

To avoid that, we can mark a dependency as optional using `@Optional` decorator and Angular will return null if not found and won't throw an error.

```
constructor(@Optional private service: LoggerService) { }
```

5.3.3 PROVIDERS

A provider is an object that tells an injector how to obtain or create a dependency, besides classes, you can also use other values such as Boolean, string, date, and objects as dependencies. Angular DI provides the necessary APIs to make the dependency configuration flexible, so you can make those values available in DI.

In the following example, the Logger class provides a Logger instance.

```
providers: [Logger]
```

You can, however, configure a DI to use a different class or any other different value to associate with the Logger class. So, when the Logger is injected, this new value is used instead.

In fact, the class provider syntax is a shorthand expression that expands into a provider configuration, defined by the Provider interface.

Angular expands the providers value in this case into a full provider object as follows:

```
[{ provide: Logger, useClass: Logger }]
```

The expanded provider configuration is an object literal with two properties:

- The **provide** property holds the **token** that serves as the key for both locating a dependency value and configuring the injector.
- The **second** property is a provider **definition object**, which tells the injector how to create the dependency value. The provider-definition key can be one of the following:
    - **`useClass`** - this option tells Angular DI to instantiate a provided class when a dependency is injected
    - **`useFactory`** - allows you to define a function that constructs a dependency.
    - **`useValue`** - provides a static value that should be used as a dependency.

Class providers: `useClass`

The useClass provider key lets you create and return a new instance of the specified class. You can use this type of provider to substitute an alternative implementation for a common or default class. The alternative implementation can, for example, implement a different strategy, extend the default class, or emulate the behavior of the real class in a test case. In the following example, the `BetterLogger` class would be instantiated when the `Logger` dependency is requested in a component or any other class.

```
[{ provide: Logger, useClass: BetterLogger }]
```

If the alternative class providers have their own dependencies, specify both providers in the providers metadata property of the parent module or component.

```
[ UserService,
    { provide: Logger, useClass: EvenBetterLogger }]
```

In this example, `EvenBetterLogger` displays the username in the log message. This logger gets the user from an injected `UserService` instance.

```
@Injectable()
export class EvenBetterLogger extends Logger {
  constructor(private userService: UserService) { super(); }

  override log(message: string) {
    const name = this.userService.user.name;
    super.log(`Message to ${name}: ${message}`);
  }
}
```

Value providers: `useValue`

The useValue key lets you associate a fixed value with a DI token. Use this technique to provide runtime configuration constants such as website base addresses and feature flags. You can also use a value provider in a unit test to provide mock data in place of a production data service.

- **Using string token**

Register the dependency using a string token like the following:

```
providers: [{ provide: "app.config", useValue: CONFIG }]
```

To use the value in a component or service (or any class in Angular), use the decorator `@Inject` with the corresponding string token:

```
constructor(@Inject("app.config") config: AppConfig) {
    this.title = config.title;
  }
```

- **Using `InjectionToken` object**

The following example defines a token, `APP_CONFIG` of the type `InjectionToken`.

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

export const APP_CONFIG = new InjectionToken<AppConfig>('app.config');
```

Next, register the dependency provider in the component using the `InjectionToken` object of `APP_CONFIG`.

```
providers: [{ provide: APP_CONFIG, useValue: CONFIG }]
```

Now, inject the configuration object into the constructor with `@Inject` parameter decorator.

```
constructor(@Inject(APP_CONFIG) config: AppConfig) {
    this.title = config.title;
  }
```

Factory providers: `useFactory`

The useFactory provider key lets you create a dependency object by calling a factory function. With this approach you can create a dynamic value based on information available in the DI and elsewhere in the app. You can also switch between providers at runtime.

In the following example, only authorized users should see secret heroes in the `HeroService`. Authorization can change during the course of a single application session, as when a different user logs in.

To keep security-sensitive information in `UserService` and out of `HeroService`, give the `HeroService` constructor a Boolean flag to control display of secret heroes.

To implement the `isAuthorized` flag, use a factory provider to create a new logger instance for `HeroService`.

```
const heroServiceFactory =(logger: Logger, userService: UserService) =>
{
    return new HeroService(logger, userService.user.isAuthorized);
}
```

The factory function has access to `UserService`. You inject both `Logger` and `UserService` into the factory provider so the injector can pass them along to the factory function.

```
providers: [
    {
        provide: HeroService,
        useFactory: heroServiceFactory,
        deps: [Logger, UserService]
    };
]
```

- The **`useFactory`** field specifies that the provider is a factory function whose implementation is `heroServiceFactory`.
- The **`deps`** property is an array of provider tokens. The `Logger` and `UserService` classes serve as tokens for their own class providers. The injector resolves these tokens and injects the corresponding services into the matching `heroServiceFactory` factory function parameters.

Module 6: Pipes

6.1 OVERVIEW

In this module, you will learn how to use pipes to transform strings, currency amounts, dates, and other data for display, first this module will introduce built-in pipes and then how to implement custom pipes and finally see the difference between pure and impure pipes.

6.2 PIPE

Pipes are transformers that accept an input value and return a transformed value. They do not alter the data but change how they appear to the user. Pipes are useful because you can use them throughout your application in HTML views or TS classes, while only declaring each pipe once.

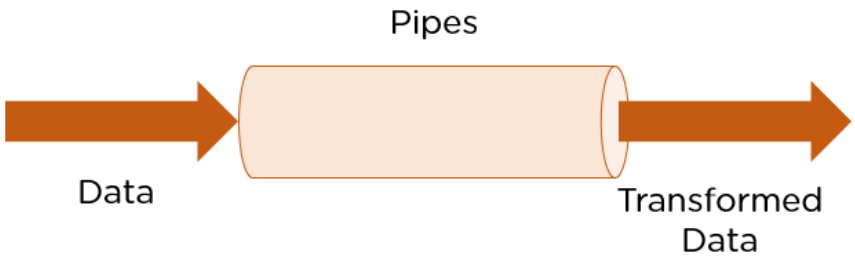


Figure 20 Pipes

Technically, pipes are simple functions designed to accept an input value, process, and return a transformed value as the output. Angular supports many built-in pipes. However, you can also create custom pipes that suit your requirements.

```
{{ value | pipe1 | pipe2:param1:param2:param3:… | … }}
```

- Pipes are defined using the pipe "|" symbol.
- Pipes can be chained with other pipes.
- Pipes can be provided with optional arguments by using the colon (:) sign.

6.3 BUILT-IN PIPES

As mentioned above, Angular provides several built-in pipes to beautify the data being shown on the user interface:

6.3.1 LowerCasePipe

Transforms text to all lower case.

Expression: `{{ value_expression | lowercase }}`

Example:

```
{{"Some text" | lowercase}}              // output is 'some text'
```

6.3.2 UpperCasePipe

Transforms text to all upper case.

Expression: `{{ value_expression | uppercase }}`

6.3.3 TitleCasePipe

Transforms text to title case. Capitalizes the first letter of each word and transforms the rest of the word to lower case. Words are delimited by any whitespace character, such as a space, tab, or line-feed character.

Expression: `{{ value_expression | titlecase }}`

Example:

```
{{"some text" | titlecase}}              // output is 'Some Text'
```

6.3.4 DecimalPipe

Formats a value according to digit options and locale rules. Locale determines group sizing and separator, decimal point character, and other locale-specific configurations.

Expression:

`{{ value_expression | number [ : digitsInfo [ : locale ] ] }}`

- `digitsInfo`: The value's decimal representation is specified by the `digitsInfo` parameter, written in the following format: `{a}.{b}-{c}`
  - a : minimum number of integer digits (default 1)
  - b : minimum digits after fraction (default 0)
  - c : maximum digits after fraction (default 3)
- `locale`: format a value according to locale rules. Locale determines group sizing and separator, decimal point character, and other locale-specific configurations. When not supplied, uses the value of `LOCALE_ID`, which is `en-US` by default.

Example:

```
{{ pi | number:'2.5-5' }}     // output is '03.14159'
```

6.3.5 CurrencyPipe

Transforms a number to a currency string, formatted according to locale rules that determine group sizing and separator, decimal-point character, and other locale-specific configurations.

Expression:

`{{value_expression | currency[:currencyCode[:display[:digitsInfo[:locale]]]] }}`

- `currencyCode:` The ISO 4217 currency code, such as USD for the US dollar and EUR for the euro.
- `display:` The format of the currency indicator, one of the following:
  - ***`code`***: Show the code (such as USD).
  - ***`symbol`*** (default): Show the symbol (such as $).
  - ***`symbol-narrow`***: Use the narrow symbol for locales that have two symbols for their currency. For example, the Canadian dollar CAD has the symbol CA$ and the symbol-narrow $. If the locale has no narrow symbol, uses the standard symbol for the locale.
- `digitsInfo` : (Same as for `DecimalPipe`)
- `locale`: (Same as for `DecimalPipe`)

Example:

```
{{0.259 | currency:'EUR':'code'}}              // output is 'EUR 0.26 '
{{1700.3495 | currency:'EUR':'symbol':'5.2-2'}} // output is '€ 01,700.35'
```

6.3.6 DatePipe

Formats a date value according to locale rules.

Expression:

`{{ value_expression | date [ : format [ : timezone [ : locale ]] ] }}`

- `locale` : When not supplied, undefined is the default
- `timezone` : When timezone not supplied, undefined is the default
- `format:` The format of the date to show (Default is *`mediumDate`*)

Example:

```
{{ dateObj | date }}              // output is 'Jun 15, 2015'
{{ dateObj | date:'medium' }}   // output is 'Jun 15, 2015, 9:43:11 PM'
{{ dateObj | date:'shortTime' }} // output is '9:43 PM'
{{ dateObj | date:'mm:ss' }}     // output is '43:11'
```

Pre-defined date format options:

Option	Equivalent to	Examples (given in `en-US` locale)
`'short'`	`'M/d/yy, h:mm a'`	`6/15/15, 9:03 AM`
`'medium'`	`'MMM d, y, h:mm:ss a'`	`Jun 15, 2015, 9:03:01 AM`
`'long'`	`'MMMM d, y, h:mm:ss a z'`	`June 15, 2015 at 9:03:01 AM GMT+1`
`'full'`	`'EEEE, MMMM d, y, h:mm:ss a zzzz'`	`Monday, June 15, 2015 at 9:03:01 AM GMT+01:00`
`'shortDate'`	`'M/d/yy'`	`6/15/15`
`'mediumDate'`	`'MMM d, y'`	`Jun 15, 2015`
`'longDate'`	`'MMMM d, y'`	`June 15, 2015`
`'fullDate'`	`'EEEE, MMMM d, y'`	`Monday, June 15, 2015`
`'shortTime'`	`'h:mm a'`	`9:03 AM`
`'mediumTime'`	`'h:mm:ss a'`	`9:03:01 AM`
`'longTime'`	`'h:mm:ss a z'`	`9:03:01 AM GMT+1`
`'fullTime'`	`'h:mm:ss a zzzz'`	`9:03:01 AM GMT+01:00`

Figure 21 Predefined date formats

6.4 USAGE FROM CODE

You can also use pipes from TS classes, following these steps

a. Import the pipe class

Built-in pipes are found in *@angular/common* package

```
import {DecimalPipe} from "@angular/common"
```

b. Inject it in the constructor

Using Angular's DI you can provide an instance of the pipe through injecting it in the constructor of the component (or any class) in-need

```
export class MyComponent {
    constructor(private pipe:DecimalPipe){}
}
```

c. Use the pipe

Each pipe class has a method *transform* that takes an input and extra parameters (depending on the pipe)

```
var formattedNumber = this.pipe.transform(12,'4.1-4');
```

6.5 CUSTOM PIPES

Angular makes provision to create custom pipes that convert the data in the format that you desire. Angular Pipes are TypeScript classes with the `@Pipe` decorator. The decorator has a name property in its metadata that specifies the Pipe and how and where it is used.

Bellow, is the code that Angular has for the uppercase pipe :

```
@Pipe({
    name: 'uppercase'
})
export class UpperCasePipe implements PipeTransform {

    transform(value:string): string {
        if (!value) return value;
        if (typeof(value) !== 'string') {
            throw InvalidPipeArgumentError(UpperCasePipe, value)
        }
        return value.toUpperCase()
    }

}
```

Pipe implements the *PipeTransform* interface. As the name suggests, it receives the value and transforms it into the desired format with the help of a *transform* method.

Here are the general steps to create a custom pipe:

a. Create a TypeScript Class with an export keyword.

b. Decorate it with the *@Pipe* decorator and pass the name property to it.

c. Implement the pipe transform interface in the class.

d. Implement the transform method imposed due to the interface.

e. Return the transformed data with the pipe.

f. Add this pipe class to the declarations array of the module where you want to use it.

Alternatively, you can use the following command from the Angular CLI :

```
ng generate pipe <name>
```

Once run a file *<name>.pipe.ts* will be created, containing:

```
@Pipe({
    name: '<name>'
})
export class DemopipePipe implements PipeTransform {

    transform(value: any, ...args: any[]): any {
        return null;
    }
}
```

As you can see, the *PipeTransform* interface and the *transform* method have been created, the left to do is to implement the transform method according to the purpose of the pipe.

Once implemented you can use it from HTML views:

```
<h3>
  The reverse of string is {{ property | <name>}}
</h3>
```

6.6 PURE VS IMPURE

Pure functions and impure functions are two common terms used in JavaScript. On the surface level, both functions look identical.

However, if we look closer, there are some major differences between them. As developers, it is essential to understand the similarities and differences of these functions to get the most out of them.

Pure functions

In simple terms, pure functions do not have an internal state. Therefore, all operations performed in pure functions are not affected by their state. As a result, the same input parameters will give the same deterministic output regardless of how many times you run the function.

To get a better understanding, let's consider the following example:

```
function add(a,b) {
    return a + b
  }
```

This example contains a simple `add()` function, giving a=5 and b=4 which gives 9 as the output. It is a very predictable output, and it does not depend on any external code. This makes the `add()` function a pure function.

If a function is declared pure and does not have a state, it can share many instances inside a class. Also, it is advised to avoid mutations inside pure functions.

Impure functions

An impure function is a function that contains one or more side effects. It mutates data outside of its lexical scope and does not predictably produce the same output for the same input.

For example, consider the following code snippet:

```
var addNew = 0;

function add(a,b){
  addNew =1;
  return a + b + addNew
}
```

In the above example, there is a variable named `addNew`, and it is declared outside of the `add` function. But the state of that variable is changed inside the `add` function. So, the `add` function has a side effect on a variable outside of its scope and is therefore considered an impure function.

6.6.1 Pure pipes

These pipes use pure functions. As a result of this, the pipe doesn't use any internal state and the output remains the same as long as the parameters passed remain the same. Angular calls the pipe only when it detects a change in the parameters being passed. A single instance of the pure pipe is used throughout all components.

6.6.2 IMPURE PIPES

An impure pipe in Angular is called for every change detection cycle regardless of the change in the input fields. Multiple pipe instances are created for these pipes and the inputs passed to these pipes are mutable.

6.6.3 Angular Change Detection

Change detection is the process through which Angular checks to see whether your application state has changed, and if any DOM needs to be updated. At a high level, Angular walks your components from top to bottom, looking for changes. Angular runs its change detection mechanism periodically so that changes to the data model are reflected in an application's view. Change detection can be triggered either manually or through an asynchronous event (for example, a user interaction or an XMLHttpRequest completion).

Change detection is a highly optimized, Angular try to minimize the change detection for that reason:

- Pipe transformations are **pure** by default
- Angular does not re-render the view when pipe inputs have not changed
- Most built-in pipes are pure by default (`DatePipe`, `LowerCasePipe`, `CurrencyPipe`…)

6.6.4 Implementing impure pipes

Although by default pipes are pure, you can specify impure pipes using the `pure` property in the `@Pipe` decorator as shown below:

```
@Pipe({
    name: '<name>',
    pure : true/false  //Set to false to have an impure pipe
  })
```

Module 7: Angular Routing

7.1 OVERVIEW

In a single-page app, you change what the user sees by showing or hiding portions of the display that correspond to particular components, rather than going out to the server to get a new page.

To handle the navigation from one view to the another, we use the Angular Router. The Router enables navigation by interpreting a browser URL as an instruction to change the view.

This module will introduce the client-side routing and what differentiate it with server-side routing, and then will guide you to setup routing using Angular router module, navigate between routes and finally protecting routes from unauthorized access using route guards.

7.2 CLIENT-SIDE ROUTING

In traditional applications built with Server-Side Routing when you change the URL in your browser, the browser makes a request to the server to return some HTML which it will display.

Figure 22 Server-side routing

However, we want to implement something called *Client-Side Routing*. When the URL changes in the browser we want our local application that's running in the browser (the *client*) to handle the change, we don't want the request sent to the server.

When we first navigate to a new site the server returns the html, JavaScript and CSS needed to render that page. All further changes to the URL are handled locally by the client application. Typically, the client application will make one or more API requests to get the information it needs to show the new page.

There is only ever a *single page* returned from the server, all further modifications of the page are handled by the *client* and that's why it's called a *Single Page Application*.

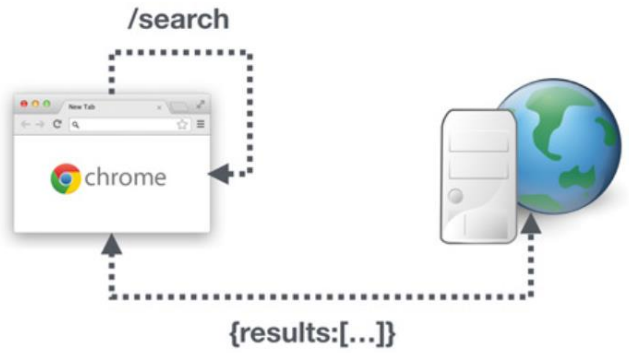


Figure 23 Client-side routing

7.3 ROUTER CONFIGURATION

The Angular's `RouterModule` is a module that provides in-app navigation among views defined in an application. It enables developers to build Single Page Applications with multiple views and allow navigation between these views.

7.3.1 Defining routes

Before we can add the `RouterModule`, we need to define the routes of our application. So, we will first look at how to define the routes. We will then come back to seeing how to import and add the `RouterModule`.

A route definition is an object of type **`Route`**, each route typically has two properties.

- The first property, **`path`**, is a string that specifies the URL path for the route.
- The second property, **`component`**, is a string that specifies what component your application should display for that path.

Example:

```
const appRoutes: Routes = [
    {
        path: "home",
        component: HomeComponent
    },
    {
        path: "login",
        component: LoginComponent
    }
];
```

7.3.2 Registering the routes with RouterModule

We will define a separate routes module file, `app-routes.module.ts`, instead of defining it in the same `app.module.ts`. This is generally good practice, as you want to keep it separate and modular, even if you only have a few routes initially. While we are just defining a separate module for our routes, it would eventually make sense for us to define a separate module and routes for each feature.

We could choose to manually create our new module and hook it up to the main `AppModule`, or let the Angular CLI do it for us, by running:

```
ng generate module app-routes --module=app
```

This would generate an `app-routes.module.ts` file in the main `app` folder. Our final `app-routing.module.ts` might look something like the following:

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';

const appRoutes: Routes = [
     { path: "home", component: HomeComponent},
     { path: "login", component: LoginComponent}
];

@NgModule({
    imports: [
        RouterModule.forRoot(appRoutes),
    ],
})
export class AppRoutesModule {
}
```

`AppRoutesModule` (annotated with `@NgModule`) simply imports the `RouterModule` and then exports it so that all modules get access to it (which we will use in a bit). we mark it for the root module by calling the *`forRoot`* method on it, with the routes we are defining.

The routes we pass to the *`forRoot`* method are nothing but an array of `Routes`. Each route is simply a configuration that defines the path for the route, as well as the component to be loaded when the route is loaded. We defined two routes, one for each of the components.

Next, we just need to hook up this module to our main module, by modifying the `app.module.ts` file as follows:

```
import { AppRoutesModule } from './app-routes.module';

@NgModule({
     declarations: [/** skipping for brevity **/],
     imports: [
       // …
        AppRoutesModule,
      ],
      providers: [/** skipping for brevity **/],
      bootstrap: [AppComponent]
    })
export class AppModule { }
```

7.3.3 Displaying route content

The last thing we need to do to get our routing app up and running is to tell Angular where to load the components when a certain route or path is matched. If you consider what we have done so far, we have defined the base for our routing and set up the module and the routes.

The final thing we do is to mark out where Angular is to load the components, and we do that by using the `RouterOutlet` element that is made available as part of the `RouterModule`. We will change the `src/app.component.html` file as follows:

```
<div>
    <span><a href="/login">Login</a></span>
    <span><a href="/home">Home</a></span>
</div>
<router-outlet></router-outlet>
```

7.3.4 Default route

If we take the previous example and run it on *`http://localhost:4200`* in our browser, we are treated with an empty page. Similarly, if we try to navigate to a URL that does not exist, and that's because the URL : *`http://localhost:4200`* does not match any routes (*`/login`* or *`/home`*).

To resolve that we need to add a default route:

```
const appRoutes: Routes = [
     { path: "", redirectTo: "home",  pathMath: "full"},
     { path: "home", component: HomeComponent},
     { path: "login", component: LoginComponent}
];
```

We have added one more entry to our routes array. Here, we match the empty path and ask Angular to redirect us to the `home` route.

Note that for any path, instead of asking Angular to use a component, we can **`redirectTo`** another already defined path as well. Also note the **`pathMatch`** key, which is set as **`full`**. This ensures that only if the remaining path matches the empty string, we redirect to the given route.

⇒ The default `pathMatch` is **`prefix`**, which would check if a URL starts with the given path. If we had the first route as the default matching one, and we forgot to add `pathMatch: full`, then every URL would match and redirect to the `home` path. Thus, both the ordering of routes as well as the `pathMatch` value is important.

7.3.5 Catch all routes (Wildcard)

The final piece we will see is how to handle if the user types in a wrong URL, or we end up having bad links in our application. It is always useful to have a catch-all route that leads to a "Page not found" page or a redirect to some other page. Let's see how we can have such a capability in our application. Again, we will only modify the `app-routes.module.ts` file:

```
const appRoutes: Routes = [
     { path: "", redirectTo: "home",  pathMath: "full"},
     { path: "home", component: HomeComponent},
     { path: "login", component: LoginComponent},
     { path: "**", redirectTo: "home"}
];
```

Our catch-all route is added by matching the path `**`. On matching this route, we then have the option of loading a component (like the other routes). Alternatively, we can redirect again, as we have done here, to another route. We redirect to the `/home` route in case we can't match the URL.

⇒ Notice that the wildcard route is placed at the end of the array. **The order of your routes is important**, as Angular applies routes in order and uses the first match it finds.

7.4 COMMON ROUTE REQUIREMENTS

In this section, we will continue digging into Angular routing capabilities, and see how we might accomplish common tasks that are needed in the course of building a web application. In particular, we will focus on having and using routes with parameters both required and optional

7.4.1 Required route params

Sometimes we have to add a route where we might depend on the route itself to decide what to load. Let's say we have a blog and each article in our blog has an ID, the URLs for each blog article might look like:

```
/blog/1
/blog/2
/blog/3
and so on...
```

Now we could write a route for each article like so:

```
const routes: Routes = [
 { path: 'blog/1', component: Article1Component },
 { path: 'blog/2', component: Article2Component },
 { path: 'blog/3', component: Article3Component },
 { path: 'blog/4', component: Article4Component },
];
```

But a better solution is to have one route with one component called `ArticleComponent` and pass to the `ArticleComponent` the number part of the URL.

That's called a **parameterized** *route* and we would implement it like so:

```
const routes: Routes = [
    { path: 'blog/:id', component: ArticleComponent }
   ];
```

A path can have any number of variables as long as they all start with "`:`" and have different names.

⇒ Note that **non-parametrized** route always takes **priority** over parametrized one**,** even if it was declared first.

7.4.2 Query params

Query parameters in Angular allow for passing optional parameters across any route in the application. The key difference between query parameters and route parameters is that route parameters are essential to **determining** route, whereas query parameters are optional and does not involve in **determining** the route.

Example:

```
http://localhost:4200/articles/1?search=angular
```

7.5 NAVIGATION

Let's hook up the navigation within the application, we can navigate between components in different ways, it can be handled in the Typescript code using `Router` Service or in the HTML template using `routerLink` directive.

7.5.1 Using `routerLink`

We can control navigation by using the `routerLink` directive in the template itself, when applied to an element in a template, makes that element a link that initiates navigation to a route. Navigation opens one or more routed components in one or more `<router-outlet>` locations on the page.

```
<nav class="navbar navbar-light bg-faded">
  <ul class="nav navbar-nav">
    <li class="nav-item active">
      <a class="nav-link"  [routerLink]="/home">Home</a>
    </li>
    <li class="nav-item">
      <a class="nav-link"  [routerLink]="/login">Login</a>
    </li>
  </ul>
</nav>
```

You can also use dynamic values to generate the link. For a dynamic link, pass an array of path segments, followed by the params for each segment. For example:

```
<a   [routerLink]="['blog', articleId]">Read article</a>
```

Relative link paths

The first segment name can be prepended with `/`, `./`, or `../`

- If the first segment begins with "`/`", the router looks up the route from the root of the app.
- If the first segment begins with "`./`", or doesn't begin with a slash, the router looks in the children of the current activated route.
- If the first segment begins with "`../`", the router goes up one level in the route tree.

Sending query params with `routerLink`

Use **`queryParams`** input for the `routerLink` directive to send the query parameters like the following:

```
<a [routerLink]="['/articles']" [queryParams]="{ search: 'angular'}">
    Angular Articles
</a>
```

7.5.2 Using `Router` service

In Angular we can also programmatically navigate via a Router service we inject into our component, like so:

```
@Component({...})
  class HeaderComponent {
    constructor(private router: Router) {}

    goHome() {
      this.router.navigate(['home']);
    }
  }
```

```
@Component({...})
  class HeaderComponent {
    constructor(private router: Router) {}

    readArticle(articleId) {
      this.router.navigate(['blog', articleId]);
    }
  }
```

Using DI system, we can inject the `Router` service into any component, and call the `navigate` method, The value we pass into the navigate function might look a bit strange, we call it a **link params array** and it's equivalent to the URL split by the `/` character into an array, this becomes a lot more useful when we start dealing with parametrized routes.

Sending query params with Router service

If you are navigating to the route imperatively using `Router.navigate`, you will pass in query parameters with `queryParams`.

```
searchArticles(filter:string) {
    this.router.navigate(
      ['/articles'],
      { queryParams: { search: filter } }
    );
  }
```

This will result in a URL that resembles:

```
http://localhost:4200/articles?search=angular
```

7.6 READ DATA

When passing parameters or query parameters in the route, we will need to read those parameters in the component in order to use their values. To do that we use something called an `ActivatedRoute`.

`ActivatedRoute` provides us with:

- Information about the current route
- Provides **`paramMap`** for parameters
- Provides **`queryParamMap`** for query parameters

We use it like the following:

```
import {ActivatedRoute} from "@angular/router";

constructor(private route: ActivatedRoute) {
    console.log(this._route.snapshot.paramMap);
}
```

Now if we navigated to `/blog/1` the id 1 would get emitted and this would get printed to the console as:

```
{id : 1}
```

7.7 NESTED NAVIGATION

Feature areas

Feature areas (or modules) are NgModules for the purpose of organizing code, as your application grows, you can organize code relevant for a specific feature. This helps apply clear boundaries for features. With feature modules, you can keep code related to a specific functionality or feature separate from other code. Delineating areas of your application helps with collaboration between developers and teams, separating directives, and managing the size of the root module.

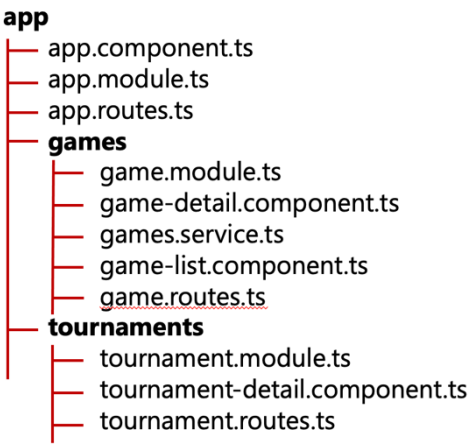


Figure 24 Feature area

A feature area is an organizational best practice, it delivers a cohesive set of functionalities focused on a specific application need such as a user workflow, routing, or forms, therefore each feature area has its own:

- Folder
- Root component
- Separate Router configuration

In order to configure router for feature modules, `RouterModule` provides two static functions: *`forRoot`* and *`forChild`* (we already used *`forRoot`* to register routes in the app module):

`forRoot`

The `forRoot` static method is the method that configures the root routing module for your app. When you call *`RouterModule.forRoot(routes),`* you are asking Angular to instantiate an instance of the Router class globally. The *`forRoot`* static method is a part of a pattern that ensures that you are using singleton classes, so *`forRoot`* needs to be called only once in the root module (`AppModule`), it:

- Make router services available for entire app
- Makes router components available in `AppModule`

`forChild`

When you are using the *`forChild`* static method, you are basically telling Angular, "There is already a `Router` instance available in the app so please just register all of these routes with that instance."

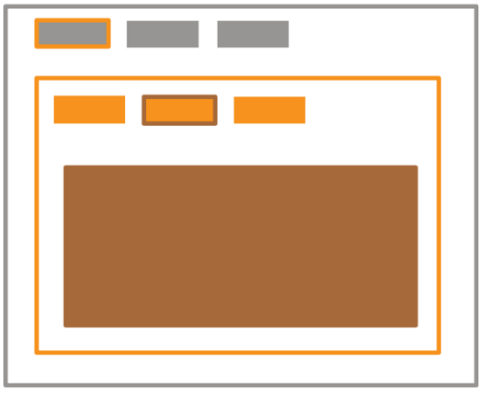
The *`forChild`* method is the method that you will call to register routes throughout your app and you will use it inside of the child, routing modules that you create. It:

- Makes router components available in the feature module
- Does **NOT** expose router services again

The *`forChild`* static method is useful because it plays a core part of Angular module functionality by allowing you to maintain separation of concerns within your app.

Multiple levels of navigation

Suppose you want to create a new feature module for user settings in your app and this feature will contain a few routes. Instead of adding these routes into the `AppRoutingModule` directly, which would eventually become untenable as your app grows, you can maintain separation of concerns within your app by using the *`forChild`* method.



http://localhost:4200/**settings**/**profile**

Figure 25 Multiple-level navigation

First, register the routes in the `UserSettingsModule`

```
const settingsRoutes: Routes = [
    {
        path: 'settings',
        component: SettingsComponent,
        children: [
            {
                path: 'profile',
                component: UserProfileComponent
            }
        ]
    }
];
@NgModule({
    imports: [RouterModule.forChild(routes)],
})
export class UserSettingsModule { }
```

Note the use of the *forChild* method above. Since you have already used the *forRoot* method, you'll just want to register your routes to the already instantiated app router.

Each route can have a property called *children* where you can define the child routes of this route.

7.8 ROUTE GUARDS

The next thing we will cover is the concept of route guards. Route guards in Angular are a way of protecting the loading or unloading of a route based on your own conditions. Route guards give you a lot of flexibility in the kinds of checks you want to add before a route opens or closes. In this section, we will deal with three in particular: a guard to prevent a route from opening, a guard to prevent a route from closing, and a guard that loads necessary data before a route is opened. We will keep the examples very simple, but these could be extended to do whatever is needed in your use case.

They make this decision by looking for a *true* or *false* return value from a class which implements the given guard interface.

There are five types of guards:

- **`CanActivate`**: Checks to see if a user can visit a route.
- **`CanActivateChild`**: Checks to see if a user can visit a child route.
- **`CanDeactivate`**: Checks to see if a user can exit a route.
- **`Resolve`**: Performs route data retrieval before route activation.
- **`CanLoad`**: Checks to see if a user can route to a module that lazy loaded.

7.8.1 `CanActivate`

Guards are implemented as services that need to be provided so we typically create them as `@Injectable` classes.

Guards return either *true* if the user can access a route or *false* if they can't.

They can also return an *Observable* or *Promise* that later on resolves to a *boolean* in case the guard can't answer the question straight away, for example it might need to call an API. Angular will keep the user waiting until the guard returns true or false.

Let's create a simple `CanActivate` guard.

```
@Injectable({
    providedIn: 'root'
})
export class AuthGuard implements CanActivate {
    constructor(private userService: UserService, private router:
Router) {
    }

    canActivate(Route: ActivatedRouteSnapshot, state:
RouterStateSnapshot) {
        if (this.userService.isAuthenticated) {
            return true
        } else {
            this.router.navigate(["/login"])
            return false
        }
    }
```

This guard returns true if the user is authenticated, in this case the navigation can complete. If it's not the case (user not authenticated) it overrides the current navigation by redirecting to the /login route to let the user authenticates.

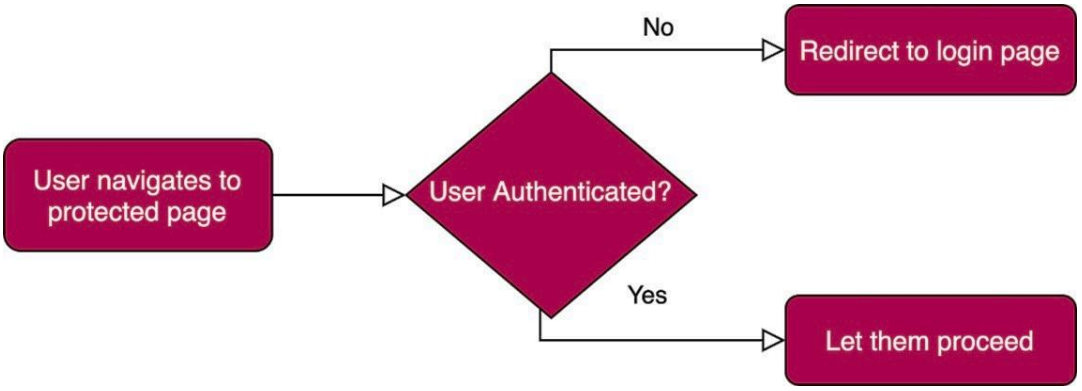


Figure 26 CanActivate route guard

Finally we need to add this guard to one or more of our routes :

```
const appRoutes: Routes = [
     { path: "", redirectTo: "home",  pathMath: "full"},
     { path: "home",
      canActivate: [AuthGuard],
      component: HomeComponent},
     { path: "login", component: LoginComponent},
     { path: "**", redirectTo: "home"}
];
```

Module 8: Working with forms

8.1 OVERVIEW

Handling user input with forms is the cornerstone of many common applications. Applications use forms to enable users to log in, to update a profile, to enter sensitive information, and to perform many other data-entry tasks.

Angular provides two different approaches to handling user input through forms: reactive and template-driven. Both capture user input events from the view, validate the user input, create a form model and data model to update, and provide a way to track changes.

This module introduces the common building blocks used by both approaches. It also summarizes the key differences between the two approaches, and demonstrates those differences in the context of setup, directives, validation and feedback.

8.2 FORM MODEL

In Angular a form is represented as a model composed of instances of `FormGroups` and `FormControls`.

8.2.1 `FormControl`

It's an interface which tracks the value and validation status for an individual form input, it implements some functionalities for accessing the value, validation status, user interactions and events.

Example:

```
const control = new FormControl<string>('some value');
console.log(control.value);     // 'some value'
```

`FormControl` takes a single generic argument, which describes the type of its value. This argument always implicitly includes null because the control can be reset. It has the following properties:

- `value`
- `dirty/pristine`: indicates if the value has been changed
- `touched/untouched`: indicates if the input has been visited
- `valid`: true/false according to passed validators
- `errors`: key/value pairs
- …

8.2.2 FormGroup

It's an interface which tracks the value and validation status a group of form controls. A `FormGroup` aggregates the values of each child `FormControl` into one object, with each control name as the key. It calculates its status by reducing the status values of its children. For example, if one of the controls in a group is invalid, the entire group becomes invalid.

When instantiating a `FormGroup`, pass in a collection of child controls as the first argument. The key for each child registers the name for the control.

Example:

```
let personInfo = new FormGroup({
    firstName: new FormControl("Johnny"),
    lastName: new FormControl("Bravo")
  });
```

Like the `FormControl` instance, `FormGroup` instances encapsulates the state of all of its inner controls, for example an instance of a FormGroup is valid only if all of its inner controls are also valid.

8.2.3 Angular Forms

In angular we can build forms in two different ways:

Template-Driven forms

Template-driven forms, as the name suggests, start with the template, and use data binding (2-way binding with `ngModel`) to get the data to and from your components. It is template-first, and allows you to drive the logic of your application via your template.

Example:

```
<form #tshirtForm="ngForm" novalidate>
    <input [(ngModel)]="tshirt.email"
            name="email"  #email="ngModel"
            required  type="email">
</form>
```

Model-Driven forms (Reactive forms)

Unlike template-driven forms in Angular, with model-driven (also called reactive) forms, you define the entire tree of Angular form control objects in your component TS code, and then bind them to native form control elements in your template. Because the component has access to the form controls as well as the backing data model, it can push data model changes into the form control and vice versa, thus reacting to changes either way.

```
this.mForm = new FormGroup({
    'size': [TShirtSize.Medium],
    'print': ['Wazup?', Validators.required],
    'email': ['e@ma.il']
  });
```

Template-Driven	Model-Driven
• FormsModule • Based on template (HTML) • ngForm, ngModel • Validation directives • Controls are referred in template	• ReactiveFormsModule • Based on model (TS) • FormGroup, FormControl • Validation functions • Controls are referred in code
• CSS-based feedback • *ngIf-based feedback	• CSS-based feedback • *ngIf-based feedback

8.3 TEMPLATE–DRIVEN

In Template Driven Forms we don't need to create `FormGroup` and `FormControl` inside the Component. Angular provides built-in directives to bind the HTML form with the model object of Component. using these directives Angular creates `FormGroup` and `FormControl` internally for us.

8.3.1 ngForm

`ngForm` directive creates a top-level `FormGroup` instance and binds it to a form to track aggregate form value and validation status. `ngForm` directive has a selector which matches the HTML form tag.

```
<form #myForm="ngForm">...</form>
```

Here `#myForm` is the local template reference variable which refers to the `ngForm` directive. One of the things the `ngForm` directive does is create a top level `FormGroup` So that we can use `#myForm` variable to access the properties of Angular Form like `value`, `valid`, `invalid` and `errors`.

8.3.2 ngModel

The `ngForm` directive doesn't automatically detect all the controls that exist inside the `<form>` tag it's linked to. We need to explicitly register each template control with the `ngForm` directive. To do so we need to do two things to each template form control:

- Add the `ngModel` directive
- Add the name attribute

```
<div class="form-group">
          <label>Title</label>
         <input type="text" class="form-control" name="title" ngModel>
</div>
```

The `NgModel` directive creates the `FormControl` instance to manage the template form control and the name attribute tells the `NgModel` directive what key to store for that `FormControl` in the parent `FormGroup`, as shown below:

Now if you run the previous example and test `myForm.value` you will get as below:

```
{
    'title': ""
}
```

8.3.3 Two-way data binding

We can use `ngModel` directive to setup two-way data binding between a template form control and a variable on our component.

So, when the user changes the value in the template form control the value of the variable on the component automatically updates and when we change the variable on the component the template form control automatically updates.

To map template control using two-way data binding, first we need to create a variable inside component with which we will map the control and then use below syntax in template control tag.

8.3.4 Form validation

To add validation to a template-driven form, you add the same validation attributes as you would with native HTML form validation. Angular uses directives to match these attributes with validator functions in the framework.

- *`required`*: Specifies whether a form field needs to be filled in before the form can be submitted.
- *`minlength`* and *`maxlength`*: Specifies the minimum and maximum length of textual data (strings).
- *`min`* and *`max`*: Specifies the minimum and maximum values of numerical input types.
- *`type`*: Specifies whether the data needs to be a number, an email address, or some other specific preset type.
- *`pattern`*: Specifies a regular expression that defines a pattern the entered data needs to follow.

```
<input type="text" id="name" name="name" class="form-control"
      required minlength="4"
      [(ngModel)]="hero.name" #name="ngModel">
```

8.3.5 Form submit

Once all controls are valid, we can submit the form either using `ngSubmit` directive at form level or using simple click event binding with the submit button.

At the form level you can use as below:

```
<form [formGroup]="myForm" (ngSubmit)="addProduct(myForm.value)" >
        ...

</form>
```

Or with the (click) event of submit button, as shown below:

```
<form [formGroup]=”myForm”>
       ....
       ....
       <button type="submit" [disabled]="myForm.invalid"
(click)="addProduct(myForm.value)" class="btn btn-
primary">Submit</button>
</form>
```

8.3.6 FormsModule

Note *ngModel* and other form-related directives are not available by default, we need to explicitly import them through the *FormsModule* in our module:

```
@NgModule({
    declarations: [App],
    imports: [BrowserModule, FormsModule],
    bootstrap: [App]
})
export class AppModule {}
```

Note that simply by including this *FormsModule* in your application, Angular will now already apply a ngForm directive to every <form> HTML template element implicitly.

8.4 MODEL-DRIVEN

Using model driven forms (also called Reactive forms), we will be able to create and manipulate various form control objects directly into the component itself. because the component class has all the access to the form control structure and the data model, so we can push model values into the form controls and also be able to check for the updated values that have been changed by the end-user. The component can observe constantly against changes in the form-control state and react to them accordingly.

In Reactive forms, we represent the form as a model composed of instances of FormGroup and FormControls.

Once we have the form model in the component, we have to link it with the HTML template of our form, we do this using a number of directives provided by Angular forms

```
let mForm = new FormGroup({
    firstName: new FormControl("Johnny"),
    lastName: new FormControl("Bravo"),
    email: new FormControl("test@gmail.com"),
  });
```

8.4.1 formGroup

Firstly, we bind the `<form>` element to our top-level *`mForm`* property using the `formGroup` directive, like so:

```
<form [formGroup]="myform"> ... </form>
```

Now we've linked the *`mForm`* model to the form template we have access to our *`mForm`* model in our template.

Running this example snippet and showing *`mForm`*`.value` prints out the below:

```
{
    "firstName": "Johnny",
    "lastName": "Bravo",
    "lastName": "test@gmail.com",
}
```

Although we've linked the form element to the *`mForm`* model this doesn't automatically link each form control in the model with each form control in the template, we need to do this explicitly with the *`formControlName`* directives.

8.4.2 formControlName

We use the formControlName directive to map each form control in the template with a named form control in the model, like so:

```
<div >
  <label>Email</label>
  <input type="email"
         formControlName="email"
         >
</div>
```

8.4.3 Form submit

Once all controls are valid, we can submit the form the same way as template-driven forms, either using `ngSubmit` directive at form level or using simple click event binding with the submit button.

8.4.4 Using `FormBuilder` API

The `FormBuilder` provides syntactic sugar that shortens creating instances of a `FormControl`, FormGroup, It reduces the amount of boilerplate needed to build complex forms. It also allows us to set up the Validation rules for each of the controls.

```
constructor(fb: FormBuilder) {
}

ngOnInit() {
    this.form = this.fb.group({
        "firstName": ["", Validators.required],
        "lastName": ["", Validators.required]
    })
}
```

8.4.5 React to changes

To react to changes on a model-driven form we need to subscribe to the `valueChanges` observable on our `FormGroup` or `FormControl`, like so:

```
this.form.valueChanges.subscribe(values=>{
        console.log("from values changed")
        console.log(values)
    })
```

8.4.6 `ReactiveFormsModule`

Note that `formGroup`, `formControl` and other model-driven form related directives are not available by default, we need to explicitly import them through the `ReactiveFormsModule` in our module:

```
@NgModule({
    declarations: [App],
    imports: [BrowserModule, ReactiveFormsModule],
    bootstrap: [App]
})
export class AppModule {}
```

8.5 DATA VALIDATION

If you have used either the Forms or Reactive Forms modules, you will have noticed that every form has a couple of important properties:

- The form value, consisting of the values of all the individual form fields
- A form validity state, which is true if all the form fields are valid, and false if at least one of the forms fields is invalid

Each form field has its own set of business validation rules: the fields can be mandatory, have a minimum length of 10 characters, etc.

8.5.1 Validators

Validators are rules which an input control has to follow. If the input doesn't match the rule, then the control is said to be invalid.

We can apply validators either by adding attributes to the template or by defining them on our FormControls in our model.

Angular comes with a small set of pre-built validators to match the ones we can define via standard HTML5 attributes, namely `required`, `minlegth`, `maxlength` and pattern which we can access from the Validators module.

The first parameter of a FormControl constructor is the initial value of the control, we'll leave that as empty string. The second parameter contains either a single validator if we only want to apply one, or a list of validators if we want to apply multiple validators to a single control, we can use either an array ([ ]) or *`Validators.compose()`* function

Our model then looks something like this:

```
ngOnInit() {
    this.myform = new FormGroup({
        name: new FormGroup({
            firstName: new FormControl('', Validators.required),
            lastName: new FormControl('', Validators.required),
        }),
        email: new FormControl('', Validators.compose(
            Validators.required,
            Validators.pattern("[^ @]*@[^ @]*")
        )),
        password: new FormControl('', [
            Validators.minLength(8),
            Validators.required
        ]),
        language: new FormControl()
    });
  }
```

8.5.2 Built-in validators

Validators class provides a set of built-in validators that can be used by form controls.

```
class Validators {
    static min(min: number): ValidatorFn
    static max(max: number): ValidatorFn
    static required(control: AbstractControl<any, any>): ValidationErrors | null
    static requiredTrue(control: AbstractControl<any, any>): ValidationErrors | null
    static email(control: AbstractControl<any, any>): ValidationErrors | null
    static minLength(minLength: number): ValidatorFn
    static maxLength(maxLength: number): ValidatorFn
    static pattern(pattern: string | RegExp): ValidatorFn
    static nullValidator(control: AbstractControl<any, any>): ValidationErrors | null
    static compose(validators: ValidatorFn[]): ValidatorFn | null
    static composeAsync(validators: AsyncValidatorFn[]): AsyncValidatorFn | null
  }
```

Validator	Description
min	Requires the control's value to be greater than or equal to the provided number.
max	Requires the control's value to be less than or equal to the provided number.
required	Requires the control have a non-empty value.
requiredTrue	Requires the control's value be true. This validator is commonly used for required checkboxes.
email	Requires the control's value pass an email validation test.
minLength	Requires the length of the control's value to be greater than or equal to the provided minimum length.
maLength	Requires the length of the control's value to be less than or equal to the provided maximum length
pattern	Requires the control's value to match a regex pattern.
nullValidator	Validator that performs no operation
compose	Compose multiple validators into a single function that returns the union of the individual error maps for the provided control.
composeAsync	Compose multiple async validators into a single function that returns the union of the individual error objects for the provided control.

8.5.3 Form control state

The form control instance on our model encapsulates state about the control itself, such as if it is currently valid or if it's been touched.

Dirty and Pristine

We can get a reference to these form control instances in our template through the control's property of the form model, for example we can print out the dirty state of the email field like so:

```
<pre>Dirty? {{ myform.controls.email.dirty }}</pre>
```

dirty is true if the user has changed the value of the control.

The opposite of dirty is pristine so if we wrote:

```
<pre>Pristine? {{ myform.controls.email.pristine }}</pre>
```

This would be *true* if the user hasn't changed the value, and *false* if the user has.

Touched and Untouched

A control is said to be touched if the user focused on the control and then focused on something else. For example, by clicking into the control and then pressing tab or clicking on another control in the form.

The difference between touched and dirty is that with touched the user doesn't need to actually change the value of the input control.

```
<pre>Touched? {{ myform.controls.email.touched }}</pre>
```

touched is true of the field has been touched by the user, otherwise it's false.

The opposite of touched is the property untouched.

Valid and Invalid

We can also check the `valid` state of the control with:

```
<pre>Valid? {{ myform.controls.email.valid }}</pre>
```

valid is true of the field doesn't have any validators or if all the validators are passing.

Again the opposite of valid is invalid, so we could write:

```
<pre>Invalid? {{ myform.controls.email.invalid }}</pre>
```

This would be true if the control was invalid and false if it was valid.

8.5.4 Validation styling

Providing feedback for the user is important when it comes to forms, for instance we can use the previous control state along with directives like `ngIf`, `ngClass` or `ngStyle` to style inputs or show/hide messages.

Using `FormControl`

```
<p *ngIf="!email.valid && email.dirty">Invalid</p>
<p *ngIf="email.errors?.required && email.dirty">Required</p>
<div [class.error]="!print.valid && print.touched"></div>
```

Using `FormGroup`

```
<button type="submit"   [disabled]="!tshirtForm.valid">
        Submit
</button>
```

Using Angular validation CSS classes

Angular dynamically adds CSS classes to allow styling of form and input controls based on the state of form field.

CSS Class	Description
ng-valid	Angular will set this CSS class if the input field is valid without errors.
ng-invalid	Angular will set this CSS class if the input does not pass validations.
ng-pristine	Angular will set this CSS class if the value of form field has not been changed yet.
ng-dirty	Angular will set this CSS class if the value of form field has been changed.
ng-touched	Angular will set this CSS class if a user tabbed out from the input control.
ng-untouched	Angular will set this CSS class if a user has not tabbed out from the input control.
ng-submitted	Angular will set this CSS class if the form has been submitted.

Note that you must provide implementation for those CSS classes, example:

```
input.ng-valid.ng-dirty {
    outline: 2px solid green;
  }

input.ng-invalid.ng-dirty {
    outline: 2px solid red;
  }
```

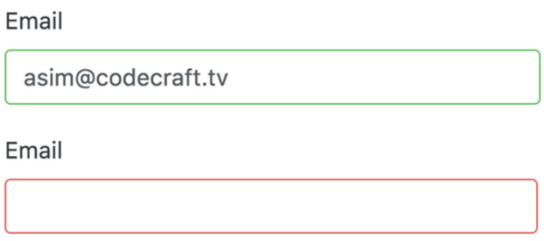


Figure 27 Form validation styling

Module 9: Reactive Programming with RxJS

9.1 OVERVIEW

Angular makes use of observables as an interface to handle a variety of common asynchronous operations, for example the HTTP module uses observables to handle AJAX requests and responses, the Router and Forms modules use observables to listen for and respond to user-input events.

In this module, we will get to know how Angular takes advantage of reactive programming paradigm the observer pattern, then we will learn how to use RxJS library to create and observables and subjects to synchronize data for multiple targets and finally how to combine and transform observables using RxJS operators.

9.2 REACTIVE PROGRAMMING

In computing, reactive programming is a declarative programming paradigm concerned with data streams and the propagation of change. With this paradigm, it's possible to express static (e.g., arrays) or dynamic (e.g., event emitters) data streams with ease, and also communicate that an inferred dependency within the associated execution model exists, which facilitates the automatic propagation of the changed data flow.

React programming is programming with asynchronous data streams that can be CREATED, CHANGED or COMBINED on the go, it focuses on responding to changes rather than asking for data, reactive programming enables the data stream to be emitted from one source called Observable and the emitted data stream to be caught by other sources called Observer through a process called subscription.

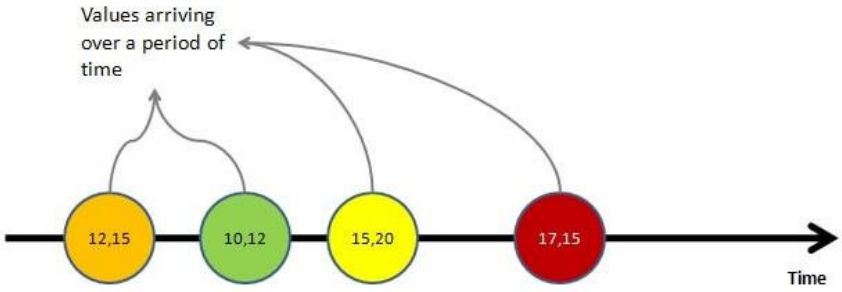


Figure 28 Data stream

This Observable / Observer pattern or simple **Observer** pattern allows sending data to other objects effectively without any change in the Subject or Observer classes, therefore simplifying complex change detection and necessary updating in the context of the programming.

Reactive programming is in the core of Angular framework, we find it almost everywhere: component's change detection, handling events, subscribing to the route's parameters, checking the status of a form, handling HTTP requests, and more.

9.3 OBSERVER PATTERN (OVERVIEW)

9.3.1 Definition

The observer design pattern – often abbreviated to observer pattern – is one of the most popular pattern templates for designing computer software. It provides a consistent way to define a one-to-one dependency between two or more objects in order to relay all changes made to a certain object as quickly and simply as possible. For this purpose, any objects that act as observer in this case can register with another object. The latter object – referred to as a subject – informs the registered observers as soon as it has changed.

9.3.2 How it functions?

The observer design pattern works with two types of actors: On the one side, there is the **subject** – i.e. the object whose status is to be observed over the long term. On the other side, there are the **observing objects** (observers) that want to be informed about any changes to the subject.

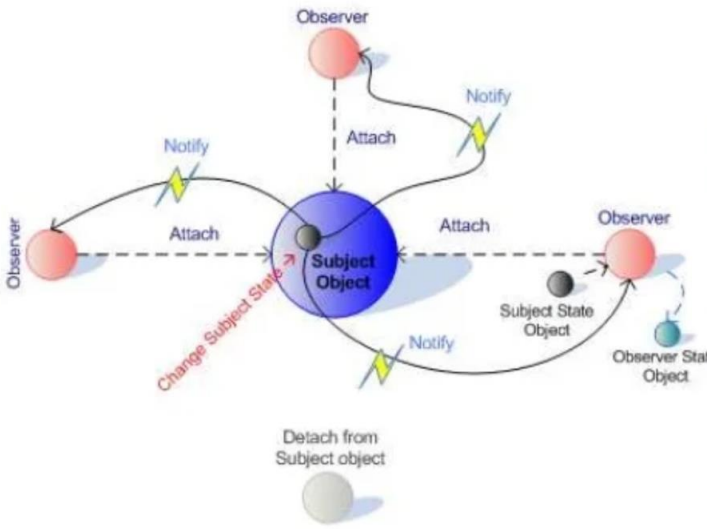


Figure 29 Observer pattern

Without using the observer pattern, the observing objects would have to ask the subject to provide status updates at regular intervals; each individual request would be associated with corresponding computing time as well as the necessary hardware resources. The fundamental idea behind the observer pattern is to centralize the task of informing within the subject. It, therefore, maintains a list which the observers can register to join. In the event of a change, the subject informs the registered observers – one after the other – without them having to take action themselves. If an automatic status update is no longer desired for a certain, observing object, it can simply be removed from the list.

9.3.3 Diagram

Observer pattern define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. **Subject** contains a list of observers to notify of any change in its state, so it should provide methods using which observers can register and unregister themselves. Subject also contain a method to notify all the observers of any change and either it can send the update while notifying the observer or it can provide another method to get the update. Observer should have a method to set the object to watch and another method that will be used by Subject to notify them of any updates.

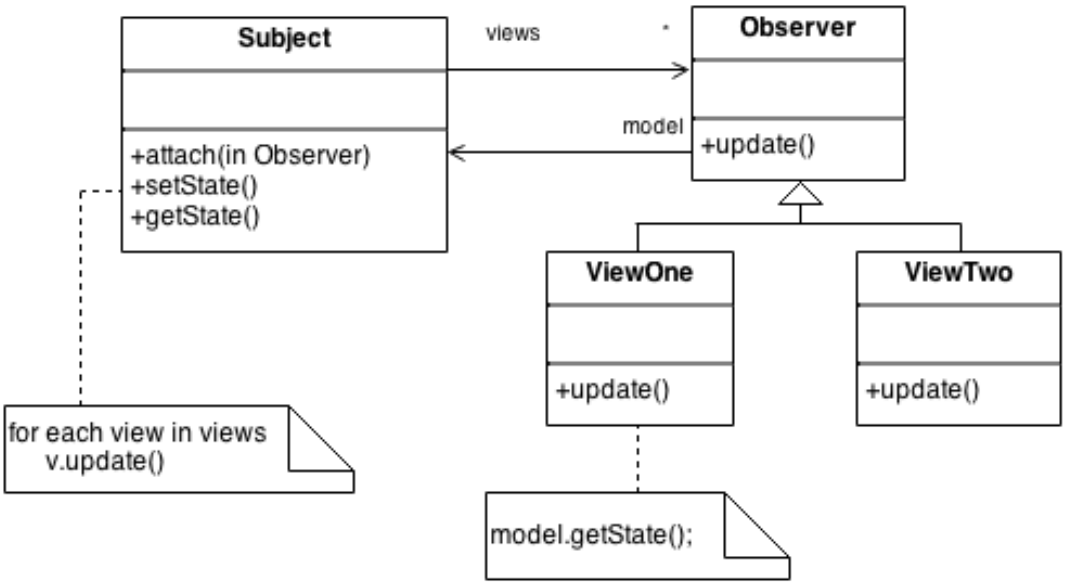


Figure 30 Observer pattern (UML)

9.4 RXJS LIBRARY

RxJS (Reactive eXtensions for JavaScript) is a library for reactive programming using Observables, to make it easier to compose asynchronous or callback-based code. RxJS provides an implementation of the Observable type, The library also provides utility functions for creating and working with observables. These utility functions can be used for:

- Converting existing code for async operations into observables
- Iterating through the values in a stream
- Mapping values to different types
- Filtering streams
- Composing multiple streams

RxJS is part of the ReactiveX, a collection of libraries for observables: RxJava, Rx.NET, RxPy, …etc.

9.4.1 Observable

`Observable` is a new primitive type provided by RxJS which implemented the observer design pattern. Observables act as a blueprint for how we want to create streams, subscribe to them, react to new values, and combine streams together to build new ones. Observables are **unicast,** each subscribed observer owns an independent execution of the observable.

Observables are created using `new` (constructor) or a creation operator, are subscribed to with an Observer, execute to deliver notifications (success, error or completion) to the Observer, and their execution may be disposed. These four aspects are all encoded in an Observable instance, but some of these aspects are related to other types, like Observer and Subscription.

RxJS is mostly useful for its *operators*, even though the `Observable` is the foundation. Operators are the essential pieces that allow complex asynchronous code to be easily composed in a declarative manner.

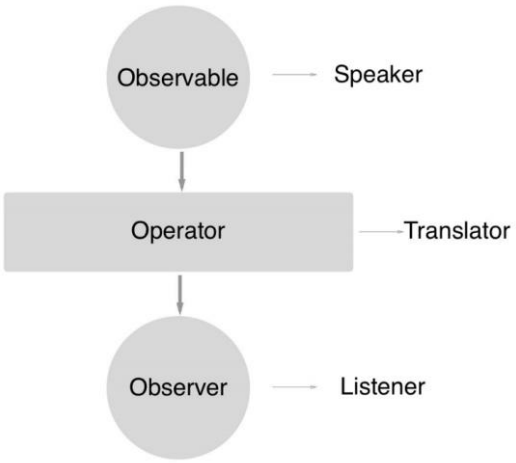


Figure 31 Observable

Observer

An Observer is a consumer of values delivered by an Observable. Observers are simply a set of callbacks, one for each type of notification delivered by the Observable: `next`, `error`, and `complete`. The following is an example of a typical Observer object:

```
const observer = {
   next: x => console.log('Observer got a next value: ' + x),
   error: err => console.error('Observer got an error: ' + err),
   complete: () => console.log('Observer got a complete notification'),
};
```

Observers are just objects with three callbacks, one for each type of notification that an Observable may deliver:

- `next`: called when observable emit the next value of the stream (observable)
- `error`: called when an error is thrown
- `complete`: called when the observable complete (close the stream)

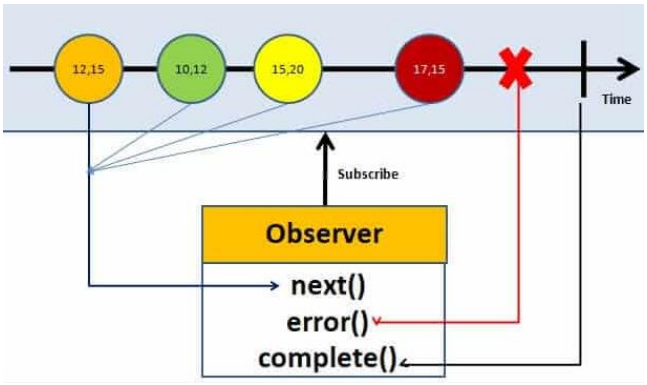


Figure 32 RxJS Observer

Subscription

A subscription is what a method in Angular that connects the observer to observable events. Whenever any change is made in these observables, a code is executed and observes the results or changes using the subscribe method.

In RxJS implementation, a Subscription is an object that represents a disposable resource, usually the execution of an Observable. A Subscription has one important method, unsubscribe, that takes no argument and just disposes the resource held by the subscription, in other words unsubscribe the observer form the observable.

```
import { Observable } from 'rxjs';

const observable = new Observable((subscriber) => {
  subscriber.next(1);
  subscriber.next(2);
  subscriber.next(3);
  setTimeout(() => {
    subscriber.next(4);
    subscriber.complete();
  }, 1000);
});
console.log('just before subscribe');
observable.subscribe({
  next(x) {
    console.log('got value ' + x);
  },
  error(err) {
    console.error('something wrong occurred: ' + err);
  },
  complete() {
    console.log('done');
  },
});
```

Example

The above is an Observable that pushes the values 1, 2, 3 immediately (synchronously) when subscribed, and the value 4 after one second has passed since the subscribe call, then completes.

Which executes as such on the console:

```
just before subscribe
got value 1
got value 2
got value 3
just after subscribe
got value 4
done
```

9.4.2 Observable VS Promise

In JavaScript, Promise object represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

Both observables and promises help us work with asynchronous functionality in JavaScript. Promises deal with one asynchronous event at a time, while observables handle a sequence of asynchronous events over a period of time.

Let's see the difference between observable and promise (observable vs promise)

Observable	**Promise**
Emit multiple values over a period of time.	Emit a single value at a time.
Are lazy: they're not executed until we subscribe to them using the `subscribe()` method.	Are not lazy: execute immediately after creation.
Have subscriptions that are cancellable using the `unsubscribe()` method, which stops the listener from receiving further values.	Are not cancellable.
Can be chained using RxJS operators, like map, filter, reduce…	Cannot be chained
next, err, complete callbacks	Resolve, reject callbacks

Now let's see code snippets / examples of a few operations defined by observables and promises.

	Observable	Promise

Creation	const obs = new Observable((observer) => {
	observer.next(10);
	}) ;
		const promise = new Promise(() =>{
		resolve(10);
		});

Transform	Obs.pipe(map(value) => value * 2);	promise.then((value) => value * 2);
Subscribe	const sub = obs.subscribe((value) => { console.log(value) });	promise.then((value) => { console.log(value) });
Unsubsribe	sub.unsubscribe();	Can't unsubscribe

With this information, often Observable is preferred over Promise because it provides the features of Promise and more. More than that RxJS provides utility that functions that helps us convert observable to promises and the other way around.

- You can convert an observable to promise by calling **`.toPromise()`**
- You can create an observable from a promise by calling **`.fromPromise()`**

9.4.3 Subject

An RxJS `Subject` is a special type of Observable that allows values to be multi-casted to many Observers. While plain Observables are unicast (each subscribed Observer owns an independent execution of the Observable), Subjects are multicast. A Subject is like an Observable, but can multicast to many Observers. Subjects are like `EventEmitters`: they maintain a registry of many listeners.

Every Subject is an Observable. Given a Subject, you can subscribe to it, providing an Observer, which will start receiving values normally. From the perspective of the Observer, it cannot tell whether the Observable execution is coming from a plain unicast Observable or a Subject.

Internally to the Subject, subscribe does not invoke a new execution that delivers values. It simply registers the given Observer in a list of Observers, similarly to how `addListener` usually works in other libraries and languages.

Every Subject is an Observer. It is an object with the methods `next(v)`, `error(e)`, and `complete()`. To feed a new value to the Subject, calling `next()` will multicast the value to the Observers registered to listen to the Subject.

```
const subject = new Subject<number>();
subject.subscribe({
  next: (v) => console.log(`observerA: ${v}`),
});
subject.subscribe({
  next: (v) => console.log(`observerB: ${v}`),
});
subject.next(1);
subject.next(2);
```

In the example above, we have two Observers attached to a Subject, and we feed some values to the Subject, the console shows the following:

```
// Logs:
// observerA: 1
// observerB: 1
// observerA: 2
// observerB: 2
```

9.4.4 BehaviorSubject

One of the variants of Subjects is the `BehaviorSubject`, which has a notion of "the current value". It stores the latest value emitted to its consumers, and whenever a new Observer subscribes, it will immediately receive the "current value" from the `BehaviorSubject`.

you can create `BehaviorSubjects` with a start value by passing along an initial value for the constructor

In the following example, the BehaviorSubject is initialized with the value 0 which the first Observer receives when it subscribes. The second Observer receives the value 2 even though it subscribed after the value 2 was sent.

```
const subject = new BehaviorSubject(0); // 0 is the initial value

subject.subscribe({
  next: (v) => console.log(`observerA: ${v}`),
});

subject.next(1);
subject.next(2);

subject.subscribe({
  next: (v) => console.log(`observerB: ${v}`),
});

subject.next(3);

// Logs
// observerA: 0
// observerA: 1
// observerA: 2
// observerB: 2
// observerA: 3
// observerB: 3
```

9.4.5 Difference between Observable, Subject and BehaviorSubject

Observable

- They are **cold**: Code gets executed when they have at least a single observer.
- Observable creates **copy of data** for each observer.
- **Uni-directional**: Observer cannot assign value to observable
- Observers get **upcoming** events only

Subject

- They are **hot**: code gets executed and value gets broadcast even if there is no observer.
- Same data get **shared** between all observers.
- **Bi-directional**: Observer can assign value to observable
- Observers get **upcoming** events only

BehaviorSubject

- They are **hot**: code gets executed and value gets broadcast even if there is no observer.
- Same data get **shared** between all observers.
- **Bi-directional**: Observer can assign value to observable
- Observers get both **previous and upcoming** events, it will replay the message stream
- **You can set initial value**: You can initialize the observable with a default value.

9.4.6 RxJS operators

Operators are the important part of RxJS. RxJS provides a huge collection of operators. There are over a 100+ operators in RxJS that you can use with observables. An operator is a pure function that takes an observable as an input and provide the output in also in the form of an observable.

How to work with operator ?

We need a `pipe()` method to work with operators. Let's see an example of `pipe()` function.

```
let obs = of(1,2,3); // an observable
obs.pipe(
   operator1(),
   operator2(),
   operator3(),
)
```

The following list sum up the most commonly used operators:

a. Map

RxJS `map()` operator is a transformation operator used to transform the items emitted by an Observable by applying a function to each item. It applies a given project function to each value emitted by the source Observable and then emits the resulting values as an Observable.

In other words, you can say that the `map()` operator passes each source value through a transformation function to get corresponding values as output.

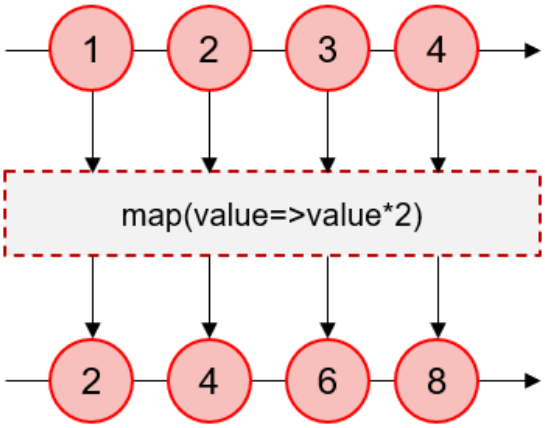


Figure 33 RxJS operators: map

```
let observable = of(1,2,3,4);
observable.pipe(
   map(value=>{
      return value*2
   })
).subscribe(value=>{
   console.log("Value:" value)
})
//logs
//Value: 2
//Value: 4
//Value: 6
//Value: 8
```

b. Filter

RxJS `filter()` operator is a filtering operator used to filter items emitted by the source observable according to the predicate function. It only emits those values that satisfy a specified predicate.

The RxJS `filter()` operator is like the well-known `Array.filter()` method. This operator takes values from the source Observable, passes them through a predicate function and only emits those values that get TRUE.

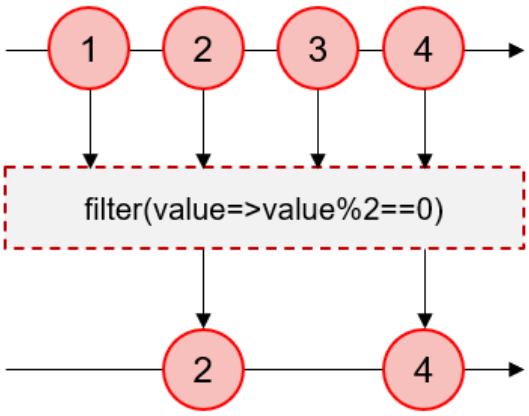


Figure 34 RxJS operators: filter

```
let observable = of(1,2,3,4);
observable.pipe(
   filter(value=>{
      return value%2==0
   })
).subscribe(value=>{
   console.log("Value:" value)
})
//logs
//Value: 2
//Value: 4
```

c. Delay

RxJS `delay()` operator is a utility operator used to delay the values emitted from the source observable according to a timeout given or by the specified delay value until a given Date.

In other words, you can say that the RxJS `delay()` operator is used to delay or time shift each item by some specified amount of milliseconds.

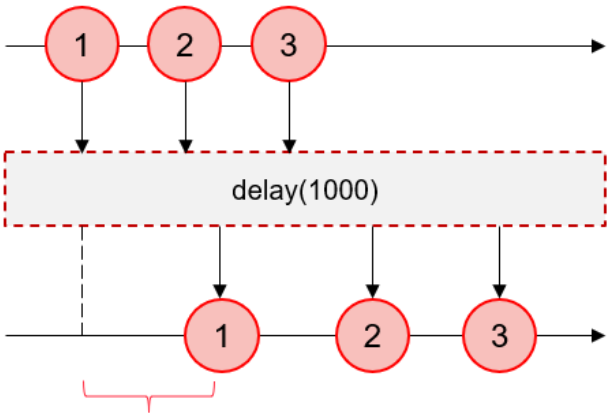


Figure 35 RxJS operators: delay

```
let observable = of(1,2,3);
observable.pipe(
   delay(1000)
).subscribe(value=>{
   console.log("Value:" value)
})
//logs (After 1second)
//Value: 1
//Value: 2
//Value: 3
```

d. Merge

The RxJS `merge()` operator is a join operator that is used to turn multiple observables into a single observable. It creates an output Observable, which concurrently emits all values from every given input Observables.

In other words, we can say that the `merge()` operator includes the multiple Observables together by blending their values into one Observable.

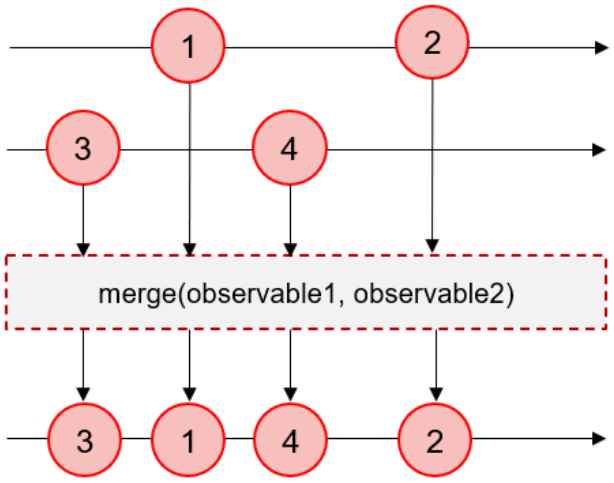


Figure 36 RxJS operators: merge

```
merge(observable1, observable2).subscribe(value=>{
    console.log("Value:" value)
})
```

e. Concat

The RxJS `concat()` operator is a join operator that creates an output Observable, which **sequentially** emits all values from the given Observable and then proceed to the next one.

Unlike the `merge()` operator, `concat()` keeps the order of observables, it does not emit the value of the next observable until the previous one complete.

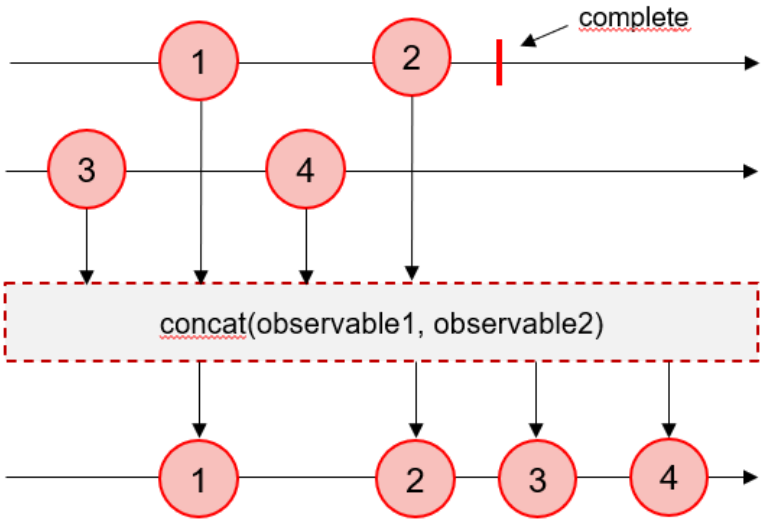


Figure 37 RxJS operators: concat

```
concat(observable1, observable2)
.subscribe(value=>{
    console.log("Value:" value)
})
```

f. catchError

RxJS `catchError()` operator is an error-handling operator used to handle and take care of catching errors on the source observable by returning a new observable or an error.

In other words, we can say that the RxJS `catchError()` operator catches errors on the observable and gracefully handles errors in an observable sequence. It handles the errors by returning a new observable or throwing an error.

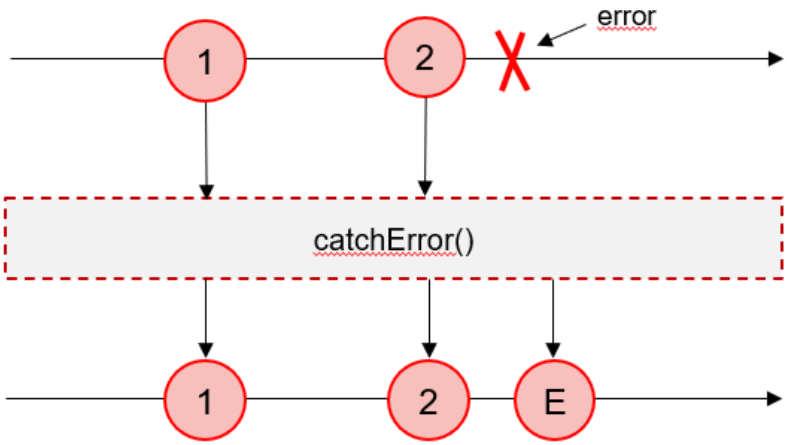


Figure 38 RxJS operators: catchError

```
observable.pipe(
    catchError(err=>{
       return of("Error happened")
    })
 ).subscribe(value=>{
    console.log("Value:" value)
 })
```

There are many other operators, according to the official documentation for categories of RxJS operators, you can find the following categories of operators:

- **Combination:** combineLatest, withLatestFrom, startWith, ...
- **Conditional:** every, iif, defaultIfEmpty, ...
- **Creation:** ajax, from, of,...
- **Error handling:** catch, catchError, retry, retryWhen, ...
- **Filtering:** debounceTime, distinctUntilChanged, take, takeUntil, ...
- **Transformation:** map, mergeMap, switchMap, mapTo, ...
- **Utility:** tap, do , delay, delayWhen, ...

Module 10: Making HTTP Requests

10.1 OVERVIEW

AJAX is the heart of single page applications. It enables the communication with the backend servers without page refresh. As Angular applications are also single page applications, it provides rich built-in support for client to server communication.

The module will guide you how to setup the HTTP client service for requesting and retrieving data from remote HTTP services and how to use the different methods the http client provides and finally introduce interceptors that are useful for intercepting requests and responses for repetitive tasks such as logging, authenticating, ...etc.

10.2 HTTP CLIENT

Most front-end applications need to communicate with a server over the HTTP protocol, to download or upload data and access other back-end services. Angular provides a client HTTP API for Angular applications, the `HttpClient` service class in `@angular/common/http`.

The HTTP client service offers the following major features:

- The ability to request typed response objects
- Streamlined error handling
- Testability features
- Request and response interception

10.2.1 Setup

Import `HttpClientModule`

```
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  imports: [
    BrowserModule,
    // import HttpClientModule after BrowserModule.
    HttpClientModule,
  ],
  declarations: [
    AppComponent,
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule {}
```

Before you can use `HttpClient`, you need to import the Angular `HttpClientModule`. Most apps do so in the root `AppModule`.

Inject `HttpClient`

You can then inject the `HttpClient` service as a dependency of an application class, as shown in the following example.

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

@Injectable()
export class ConfigService {
  constructor(private http: HttpClient) { }
}
```

10.2.2 Request data from a server

`HttpClientModule` provides `HttpClient` service that contains a set of methods that corresponds to HTTP verbs (`get`, `post`, `put`, `path`, `delete`, `head`), those methods allow to perform client-server requests and handle responses.

`HttpClient` is RxJS Observable-based, meaning we must subscribe to the return value of each method to have the response.

Each method in `HttpClient` have this signature

```
VERB<TResponse>(url: string, body: any, options?: RequestOptions): Observable<TResponse>

//Example
// post<T>()(url:string, body:any,options?: RequestOptions): Observable<TResponse>
// get<T>()(url:string, options?: RequestOptions): Observable<TResponse>
```

⇒ The method ***get*** is an exception, it doesn't support a `body` parameter

The *options* object is of type `RequestOptions` that is used to configure the request:

```
options: {
    headers?: HttpHeaders | {[header: string]: string | string[]},
    observe?: 'body' | 'events' | 'response',
    params?: HttpParams,
    reportProgress?: boolean,
    responseType?: 'arraybuffer'|'blob'|'json'|'text',
    withCredentials?: boolean,
  }
```

We use options to specify request headers (authorization for example), setting the response type of the request, sending query parameters.

Example:

```
configUrl = 'assets/config.json';

getConfig() {
  return this.http.get<Config>(this.configUrl);
}
```

The example conforms to the best practices for creating scalable solutions by defining a re-usable injectable service to perform the data-handling functionality. In addition to fetching data, the service can post-process the data, add error handling, and add retry logic.

The golden rule

Always separate client-server
communication from your
***Components**, use **Services** instead*

Because the service method returns an Observable of configuration data, the component subscribes to the method's return value. The subscription callback performs minimal post-processing. It copies the data fields into the component's config object, which is data-bound in the component template for display.

```
showConfig() {
    this.configService.getConfig()
      .subscribe((data: Config) => this.config = {
          heroesUrl: data.heroesUrl,
          textfile:  data.textfile,
          date: data.date,
      });
  }
```

For all HttpClient methods, the method doesn't begin its HTTP request until you call `subscribe()` on the observable the method returns.

All observables returned from `HttpClient` methods are cold by design. Execution of the HTTP request is deferred, letting you extend the observable with additional operations such as tap and catchError before anything actually happens.

Calling `subscribe()` triggers execution of the observable and causes `HttpClient` to compose and send the HTTP request to the server.

In fact, each `subscribe()` initiates a separate, independent execution of the observable. Subscribing twice results in two HTTP requests.

```
const req = http.get<Heroes>('/api/heroes');
// 0 requests made - .subscribe() not called.
req.subscribe();
// 1 request made.
req.subscribe();
// 2 requests made.
```

10.2.3 Handling request errors

If the request fails on the server, `HttpClient` returns an error object instead of a successful response.

The same service that performs your server transactions should also perform error inspection, interpretation, and resolution.

An app should give the user useful feedback when data access fails. A raw error object is not particularly useful as feedback. In addition to detecting that an error has occurred, you need to get error details and use those details to compose a user-friendly response.

Two types of errors can occur.

- The server backend might reject the request, returning an HTTP response with a status code such as 404 or 500. These are error responses.
- Something could go wrong on the client-side such as a network error that prevents the request from completing successfully or an exception thrown in an RxJS operator. These errors have status set to 0 and the error property contains a `ProgressEvent` object, whose type might provide further information.

`HttpClient` captures both kinds of errors in its `HttpErrorResponse` . Inspect that response to identify the error's cause.

The following example defines an error handler in the previously defined `ConfigService`.

```
private handleError(error: HttpErrorResponse) {
  if (error.status === 0) {
    // A client-side or network error occurred. Handle it accordingly.
    console.error('An error occurred:', error.error);
  } else {
    // The backend returned an unsuccessful response code.
    // The response body may contain clues as to what went wrong.
    console.error(
      `Backend returned code ${error.status}, body was: `,
error.error);
  }
  // Return an observable with a user-facing error message.
  return throwError(() =>
    new Error('Something bad happened; please try again later.'));
`
```

The handler returns an RxJS `ErrorObservable` with a user-friendly error message. The following code updates the `getConfig()` method, using a pipe to send all observables returned by the `HttpClient.get()` call to the error handler.

```
getConfig() {
    return this.http.get<Config>(this.configUrl)
      .pipe(
        catchError(this.handleError)
      );
  }
```

10.3 HTTP INTERCEPTORS

With interception, you declare interceptors that inspect and transform HTTP requests from your application to a server. The same interceptors can also inspect and transform a server's responses on their way back to the application. Multiple interceptors form a forward-and-backward chain of request/response handlers.

Interceptors can perform a variety of implicit tasks, from authentication to logging, in a routine, standard way, for every HTTP request/response.

Without interception, developers would have to implement these tasks explicitly for each `HttpClient` method call.

10.3.1 Write an interceptor

To implement an interceptor, declare a class that implements the `intercept()` method of the `HttpInterceptor` interface.

Here is an authentication interceptor that passes credentials through the request headers:

```
@Injectable()
export class AuthInterceptor implements HttpInterceptor {
   constructor(private auth: AuthService) {}
      intercept(req: HttpRequest<any>, next: HttpHandler):
Observable<HttpEvent<any>> {
      // Get the auth header from the service
      const authHeader = this.auth.getAuthorizationHeader();
      // Clone the request to add the new header
      const authReq = req.clone({headers: req.headers.set('Authorization',
authHeader)});
      // Pass on the cloned request instead of the original request
      return next.handle(authReq);
   }
}
```

The intercept method transforms a request into an Observable that eventually returns the HTTP response. In this sense, each interceptor is fully capable of handling the request entirely by itself.

Most interceptors inspect the request on the way in and forward the potentially altered request to the `handle()` method of the next object which implements the `HttpHandler` interface.

The next object represents the next interceptor in the chain of interceptors. The final next in the chain is the `HttpClient` backend handler that sends the request to the server and receives the server's response.

Most interceptors call `next.handle()` so that the request flows through to the next interceptor and, eventually, the backend handler. An interceptor could skip calling `next.handle(),` short-circuit the chain, and return its own Observable with an artificial server response.

10.3.2 Intercepting responses

Because interceptors can process the request and response together, they can perform tasks such as timing and logging an entire HTTP operation.

Consider the following `LoggingInterceptor`, which captures the response and logs it with the current time

```
@Injectable()
export class LoggingInterceptor implements HttpInterceptor {
   constructor(private log: LoggingService) {}

   intercept(req: HttpRequest<any>, next: HttpHandler):
Observable<HttpEvent<any>> {
      // Use RxJS operators to chain
      return next.handle(authReq).pipe(
         map((event: HttpEvent<any>) => {
            if (event instanceof HttpResponse) {
               log.info(new Date(), event)
            }
         return event;
     }));
  }
```

Since, intercept return an Observable and thanx to RxJS we can apply operators on an Observable type object, here we are chaining using the *map* operator, check if the event is an `HttpResponse`, log it and return the event itself. We could also modify the response if needed.

10.3.3 Provide the interceptor

Because interceptors are optional dependencies of the `HttpClient` service, you must provide them in the same injector or a parent of the injector that provides `HttpClient`. Interceptors provided after DI creates the HttpClient are ignored.

After importing the `HTTP_INTERCEPTORS` injection token from `@angular/common/http`, register the interceptors provider like this:

```
{ provide: HTTP_INTERCEPTORS, useClass: AuthInterceptor, multi: true }
```

Notice the multi: true option. This required setting tells Angular that HTTP_INTERCEPTORS is a token for a multi-provider that injects an array of values, rather than a single value. As a result, we can have many interceptors.

10.3.4 Interception order

Angular applies interceptors in the order that you provide them. For example, consider a situation in which you want to handle the authentication of your HTTP requests and log them before sending them to a server. To accomplish this task, you could provide an `AuthInterceptor` service and then a `LoggingInterceptor` service. Outgoing requests would flow from the `AuthInterceptor` to the `LoggingInterceptor`. Responses from these requests would flow in the other direction, from `LoggingInterceptor` back to `AuthInterceptor`.

The following is a visual representation of the process:

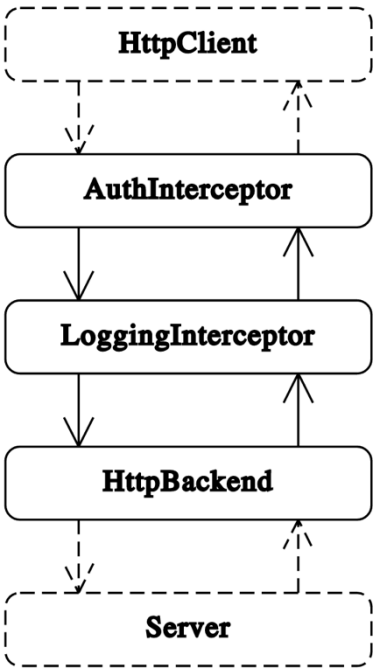


Figure 39 HTTP Interception order

The last interceptor in the process is always the `HttpBackend` that handles communication with the server.