# Introduction to Angular

# What Is Angular?

Learn what Angular is and why it is an ideal choice for developing web applications.

**We'll cover the following**

* [Angular](https://www.educative.io/courses/getting-started-with-angular/RL6ZDxG9VYY#Angular)

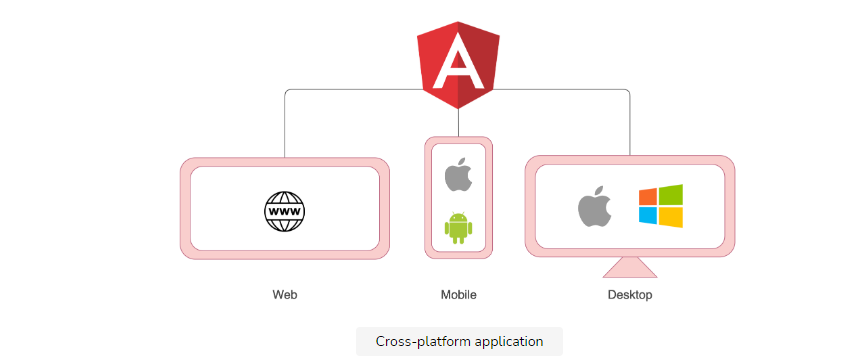
We will introduce the demo Angular application that will be built throughout the book. In this section, we will cover the following topics:

* What is Angular, and why are we looking at it?
* Some examples of the types of applications that can be built with Angular.
* The history of Angular, how it was started, and what problems it aimed to solve.
* The demo application we will be building throughout this book.

## Angular

According to the official Angular docs, Angular is a platform that makes it easy to build applications with the web. But what does that actually mean? Well, Angular is a web application framework that helps developers build web applications that can run on all platforms. This makes Angular an ideal choice for your next web application.

Cross-platform application



The official documentation describes Angular as:

Combining declarative templates, dependency injection, end-to-end tooling, and integrated best practices to solve development challenges. Angular empowers developers to build applications that live on the web, mobile, or desktop.

This perfectly describes what Angular is. Through **templates**, **dependency injection**, and **end-to-end tooling**, Angular empowers developers to build web applications and applications that are built on best practices.

# Why Use Angular?

Let's see why we should use Angular as our web application framework of choice.

**We'll cover the following**

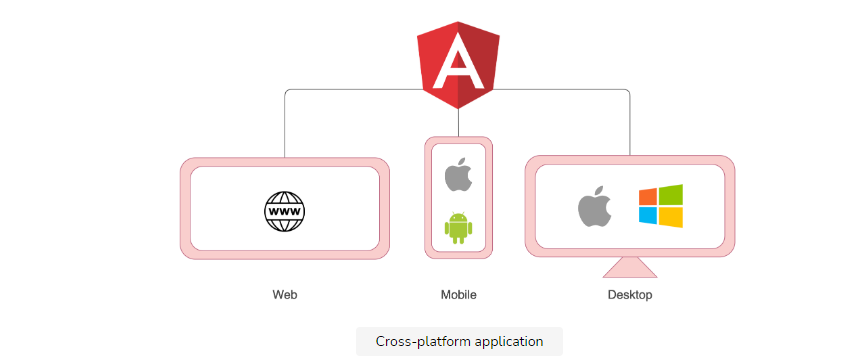
* [Cross-platform framework](https://www.educative.io/courses/getting-started-with-angular/xoM7q7gP2VB#Cross-platform-framework)
* [Supported by Google](https://www.educative.io/courses/getting-started-with-angular/xoM7q7gP2VB#Supported-by-Google)
* [Built on TypeScript](https://www.educative.io/courses/getting-started-with-angular/xoM7q7gP2VB#Built-on-TypeScript)
* [The Angular CLI](https://www.educative.io/courses/getting-started-with-angular/xoM7q7gP2VB#The-Angular-CLI)
* [Built on best practices](https://www.educative.io/courses/getting-started-with-angular/xoM7q7gP2VB#Built-on-best-practices)
* [Testing is a first-class citizen of Angular](https://www.educative.io/courses/getting-started-with-angular/xoM7q7gP2VB#Testing-is-a-first-class-citizen-of-Angular)
* [Ideal frontend framework for enterprise-level application](https://www.educative.io/courses/getting-started-with-angular/xoM7q7gP2VB#Ideal-frontend-framework-for-enterprise-level-application)
* [The Angular community](https://www.educative.io/courses/getting-started-with-angular/xoM7q7gP2VB#The-Angular-community)
* [Access to third-party libraries](https://www.educative.io/courses/getting-started-with-angular/xoM7q7gP2VB#Access-to-third-party-libraries)

## Cross-platform framework

Why should a developer learn Angular? Well, there are several reasons, but the main reason is to **create applications for all platforms**:

* Desktop
* Mobile
* Web

All these platforms are reachable for a developer who knows how to write Angular applications.



## Supported by Google

Another reason to use Angular is because Google supports it. They don’t just sponsor the development of Angular; there is a whole team behind the development of Angular. This is unlike other frameworks, which are supported by open source developers who work on fixing issues and creating new features for a framework in their spare time. Angular has a full-time team constantly working on providing maintenance and support to Angular applications

Google has also provided a [**Long Term Support**](https://angular.io/guide/releases)**(LTS)** plan for Angular. In this LTS, we can see:

* What versions of Angular they provide support for
* When older versions will no longer be supported

With Google being so transparent in terms of their support of Angular, we can be sure that it has a **long-lasting future**.

This is extremely important for large businesses looking to select the framework they are going to invest in for their next large-scale project. Being able to see that Angular has the support of a large organization like Google and that there is an LTS plan, which shows that Angular will be supported for the long term, makes the decision to pick Angular as their framework of choice a straightforward one.

## Built on TypeScript

Angular is built on TypeScript, a **superset of JavaScript**, built by Microsoft. TypeScript brings so many benefits, including type interfaces and static typing. When we create objects and variables within our code using static typing, the details of these types are known when we compile our code, and this helps to provide useful insights. This way, bugs can be found at compile time instead of runtime.

**Note:** TypeScript only checks types at compile time while the types of the response from the backend is known only at run time.

Not only does TypeScript help us write better code, but it also allows tools such as **VSCode** to provide **IntelliSense of our code**, which gives us:

* Better navigation
* Refactoring
* Autocompletion

This makes the experience of writing TypeScript far more enjoyable than JavaScript.

TypeScript can provide features that are still to be released in JavaScript. So, being built on TypeScript, Angular has access to all the latest features of the language, and all the fantastic tooling TypeScript provides.

## The Angular CLI

Another great reason for using Angular is the Angular **Command-Line Interface (CLI)**; this is a command-line tool provided by the Angular team that helps us to build and run Angular projects. The CLI comes with a wide range of features, including the following:

* Starting a project
* Creates components, services, directives, and other files using a simple command
* Runs the application in the browser
* Reloads the application after each saved change so that the latest version is displayed in the browser
* It can update a project’s dependencies (other libraries used in a project) automatically
* It can add new libraries to the project
* It can run all the unit tests in a project and the end-to-end tests

The Angular CLI is a great tool, and you’ll see as we proceed that learning it is an important part of developing Angular applications. Many other frontend frameworks don’t have CLIs, and those that do have them don’t support the features of the Angular CLI.

## Built on best practices

Angular is designed with best practices in mind. Any code generated by the CLI follows these best practices, as set out by the team from Google. By following the approach prescribed by the Angular team on how to write an Angular application, you know that the application you are building is using the best practices of modern web applications.

These best practices include the following:

* Component-based architecture
* Modularised structure
* Dependency injection
* Testing
* Readable code
* Ease of maintenance

## Testing is a first-class citizen of Angular

Testing is an important part of creating bug-free applications. Angular supports testing straight out of the box. Whenever the CLI creates a new Component or Service, it automatically creates a **Test Spec** file for the new Component or Service. The CLI can also run the tests. No longer do we have to set up test runner files; all of this is automatically managed by the CLI.

Writing and running tests are easy, so there is no reason not to have a good set of tests for your application. Angular actively encourages writing tests, and by doing this, it reduces bugs and issues that an end-user may find.

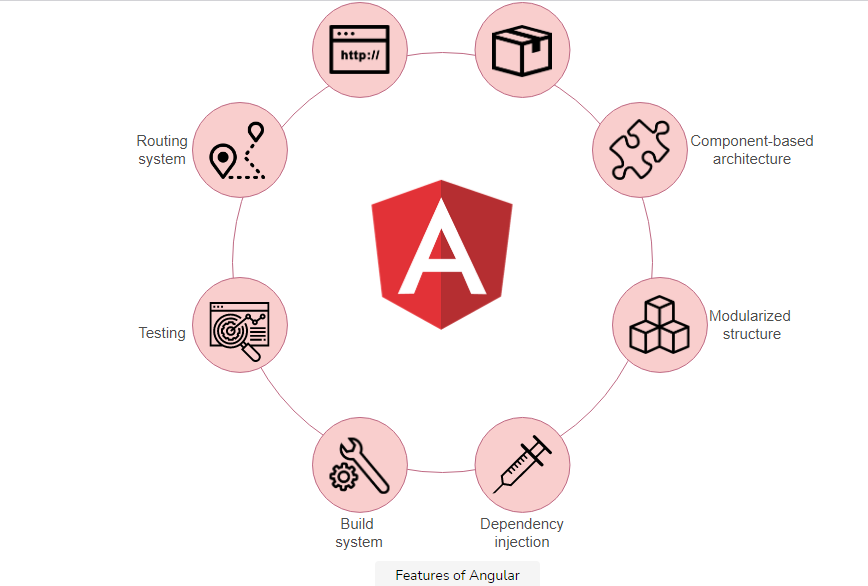
Testing in Angular

## Ideal frontend framework for enterprise-level application

There are other similar technologies out there; for example, there are many JavaScript frameworks like Angular, but Angular is **extremely well established and supported by Google**. It is an ideal frontend framework for the enterprise-level application, with all the out-of-the-box features that come with Angular, including:

* Modules
* Classes
* Components
* Unit testing

All of these are important to enterprises wanting their teams to **develop well structured, tested applications**; that are going to work the first time for their clients.



## The Angular community

Another great reason for choosing Angular as your framework of choice is the community that has grown around Angular. There are over **100,000 Angular developers**, a number that is still growing. There are **Angular meetups all around the world** where you can go and meet other Angular developers to discuss their experiences with using Angular.

There are also many conferences you can attend as an Angular developer to learn about the new features of Angular and hear talks on different approaches to working with Angular. These **conferences are all over the world** and attended by the Angular team, so you can put your questions to the team directly.

Not only are there conferences and meetups you can attend to learn about Angular, but there are also many **online resources** created by members of the Angular community that we can access to learn about Angular.

There are also **podcasts** where you hear interviews with leaders of the Angular community, including the Angular team.

There is also a huge ecosystem around Angular, teams such as [Nrwl](https://nrwl.io/" \t "_blank) with their Nx Extensions, which **allow teams to create libraries to support the large enterprise-level applications** that Nrwl specializes in. The NX tool helps the CLI create libraries within Angular projects so that large-scale teams can share code across teams.

Angular community

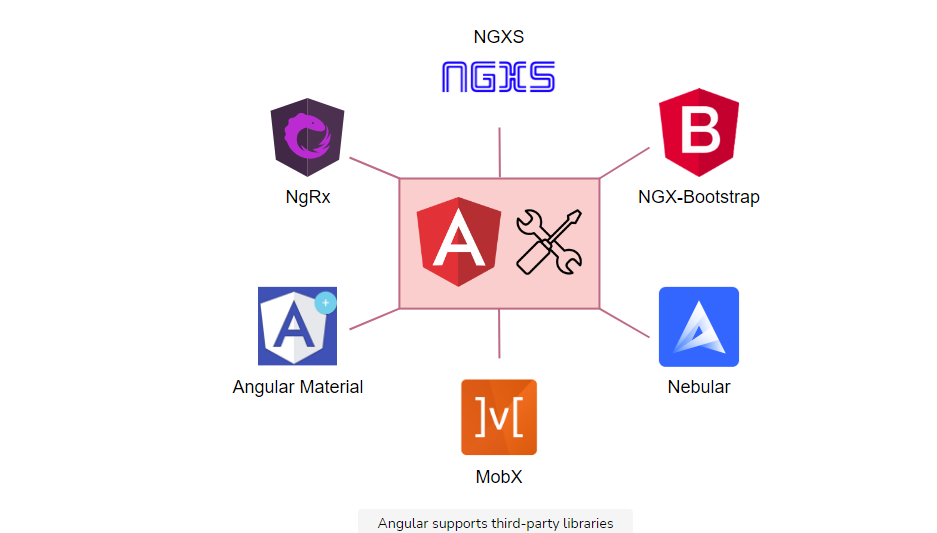
## Access to third-party libraries

The Angular community, as well as allowing us to learn from one another, is also active in building libraries and tools that work with Angular. Libraries such as [NgRx](https://ngrx.io/" \t "_blank), [NGXS](https://www.ngxs.io/), and [MobX](https://mobx.js.org/README.html" \t "_blank) all provide solutions on how to **manage state within an Angular application**. There are also **UI libraries** such as [Angular Material](https://material.angular.io/), [NGX-Bootstrap](https://valor-software.com/ngx-bootstrap/#/), and [Nebular](https://akveo.github.io/nebular/), which provide UI components you can add to your Angular applications.

There are **libraries for accessing data from cloud-based systems**, such as [Firebase](https://firebase.google.com/). So, if your application needs to connect to an existing Firebase application, there are libraries you can add that simplify working with Firebase in your application. You can also find libraries that facilitate working with [GraphQL](https://graphql.org/" \t "_blank) if that’s how your team creates their APIs.

If you want to **add tools to your Angular workflow**, the Angular community has provided tools like [Augury](https://augury.rangle.io/), which is a plugin for Chrome that debugs Angular applications.

Another great tool from the Angular community is the NX workspaces from [Nrwl](https://nrwl.io/" \t "_blank). They have created an extension to the Angular CLI that helps to create large enterprise Angular applications, where teams of developers work on the same project. The NX tool helps the CLI create libraries within Angular projects so that large-scale teams can share code across teams.



📝 **Note:** To see the types of extra resources available for Angular, look at the [Resources list](https://angular.io/resources?category=development) on the Angular website.

As you can see, there are so many reasons to use Angular. We’re already on version 11, which shows that the Angular team isn’t slowing down in terms of making Angular better and better.

# Course Project: The Client Contacts Manager Application

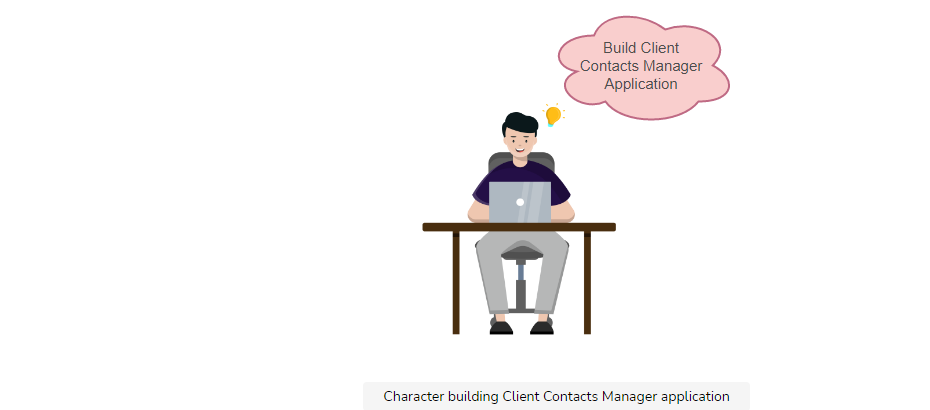
Get introduced to the features, learning outcomes, and final preview of the course project.

**We'll cover the following**

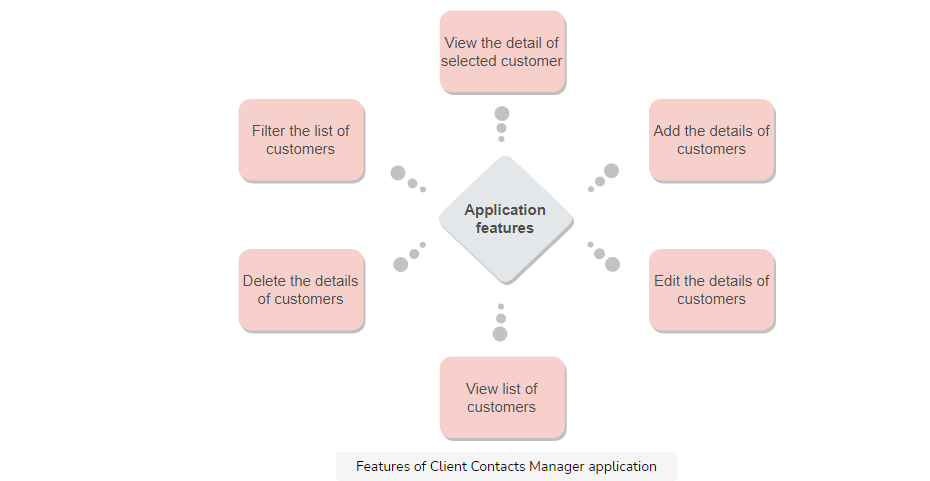
* [Project features](https://www.educative.io/courses/getting-started-with-angular/q2EGOjNORq0#Project-features)
* [Project learning outcomes](https://www.educative.io/courses/getting-started-with-angular/q2EGOjNORq0#Project-learning-outcomes)
* [Project preview](https://www.educative.io/courses/getting-started-with-angular/q2EGOjNORq0#Project-preview)

## Project features

Throughout this course, we are going to be building a **sales team contacts app for a fictional company**. The idea behind this app is to have a system that allows the fictional sales team of our fictional company to manage and access the contact details of their clients and the companies they work with.



This mini-CRM (Customer Relationship Manager) allows us to create an app that provides a number of functions so we can really explore the features of Angular.



The idea is to create two versions of the mini-CRM that the salesperson will use:

* The mobile version will be used when they are out and about talking to their customers
* The desktop app will be used when back in the office to search for all the new customers they have.

## Project learning outcomes

While building the project, you will learn how to:

* Create UI using Angular components and Angular Material
* Build up the model of the application
* Pass data throughout our app as we build up the functionality to add new customers, save their details, and see a list of all of our saved customer contacts

We will see how Angular uses TypeScript, which gives us excellent tooling and insight into our code, making the development process even easier than when we were using JavaScript.

As part of this application, we are going to use a fantastic third-party library called the [In-Memory Web API](https://github.com/angular/in-memory-web-api).

The **In-Memory Web API library** allows us to create local, in-memory storage similar to a database that we can save data to, access data from, and remove data from, all via API calls.

Using this library means we have a source of storage for our Client Contacts Manager Application, and we can make API calls to this storage system without having to set up a local database of external API.

📝 **Note:** For now, we can just focus on learning how to write Angular. Once you’ve finished reading this course and you want to create your own application to practice what you’ve learned, we highly recommend looking at the In-Memory Web API as a temporary data source for your practice applications. It’s not a replacement for a real database but is easy to use for small demo applications.

## Project preview

Below is a preview of what the final project will look like after adding Angular functionality.

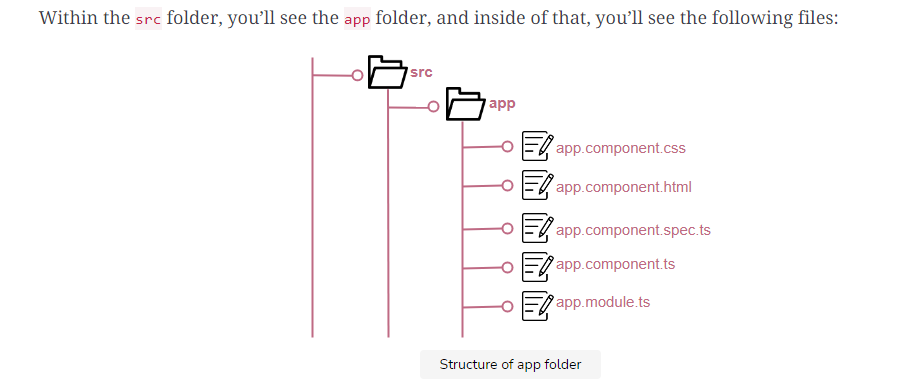
📝 **Note:** Press the **RUN** button to compile and serve the application. Once the app compiles, you can either see its output in the **Output** tab or by clicking on the URL given after Your app can be found at.



**Your app can be found at:**[https://ed-6434592358924288.educative.run](https://ed-6434592358924288.educative.run/)

You can play around with the project by searching, viewing, adding, deleting, or updating contacts and companies while focusing on its interactivity and functionality.

## Structure of app folder



# The Architecture of an Angular App

Let's explore the architecture of an Angular app and see what modules, components, and services are.

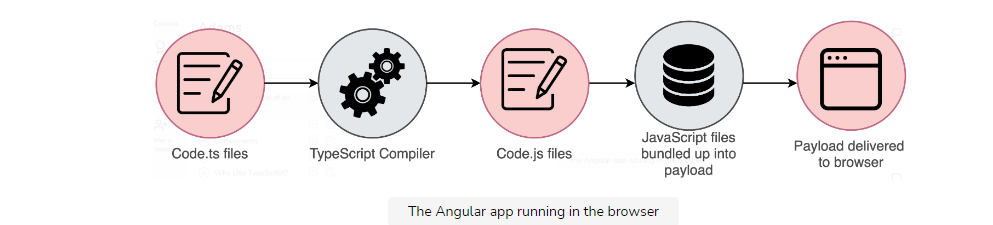
**We'll cover the following**

* [Modules](https://www.educative.io/courses/getting-started-with-angular/qAoyX486A97#Modules)
* [Components](https://www.educative.io/courses/getting-started-with-angular/qAoyX486A97#Components)
* [Services](https://www.educative.io/courses/getting-started-with-angular/qAoyX486A97#Services)
* [Angular application as a tree](https://www.educative.io/courses/getting-started-with-angular/qAoyX486A97#Angular-application-as-a-tree)
* [Quick quiz](https://www.educative.io/courses/getting-started-with-angular/qAoyX486A97#Quick-quiz)

Angular is a framework and a platform that’s **written using HTML and TypeScript**. The functionality of the application is written in TypeScript files that can be imported throughout the app as libraries, adding functionality throughout the app.

When the Angular application app is run in the browser, these TypeScript files are converted into JavaScript files, which are then bundled up into the payload that is delivered to the browser

The Angular app running in the browser



📝 **Note:** There are ways that this payload can be separated into smaller, separate files in order to keep its size down, but this is something we will briefly look at in later chapters.

There are three main parts to an Angular application, and they are as follows:

* Modules
* Components
* Services

## Modules

**Modules** are the glue that hold an application together. Each module is a single TypeScript file that references all the other files used within the application.

They allow us to group the functionality of our application together.

## Components

**Components** are the building blocks of the application. They are single pieces of functionality in our application, which are linked together under a module.

Components can have visual elements to them, which allow the user to interact with the application.

## Services

**Services** are single TypeScript classes used to access information and share it between components.

## Angular application as a tree

When you think of an Angular application, you can think of it as a tree. The module is the trunk of the tree, and the components are the branches of the tree, branching out of the module, with services being passed into components to share data throughout the application. Everything is tied together through the module, and as the complexity of your Angular application grows, the number of modules you’ll have in your application will grow.

# Modules

Get introduced to the Angular module and dissect the functionality of the main app module.

**We'll cover the following**

* [What are the modules?](https://www.educative.io/courses/getting-started-with-angular/qVxY1l4NxMD#What-are-the-modules?)
* [Structure of module](https://www.educative.io/courses/getting-started-with-angular/qVxY1l4NxMD#Structure-of-module)
  + [declarations array](https://www.educative.io/courses/getting-started-with-angular/qVxY1l4NxMD#declarations-array)
  + [imports array](https://www.educative.io/courses/getting-started-with-angular/qVxY1l4NxMD#imports-array)
  + [providers array](https://www.educative.io/courses/getting-started-with-angular/qVxY1l4NxMD#providers-array)
  + [bootstrap array](https://www.educative.io/courses/getting-started-with-angular/qVxY1l4NxMD#bootstrap-array)
  + [export array](https://www.educative.io/courses/getting-started-with-angular/qVxY1l4NxMD#export-array)
* [The functionality of NgModules](https://www.educative.io/courses/getting-started-with-angular/qVxY1l4NxMD#The-functionality-of-NgModules)

## What are the modules?

**Modules** are the glue that hold an application together. They are single TypeScript files that reference all the other files used within the application. They allow us, as Angular developers, to group the functionality of our application together.

We’ve now provided an overview of the general structure of an Angular app, so we’re going to be looking further into **modules**, or, as they are known in the Angular world, **NgModules**.

## Structure of module

To see an example of a module, open the app.module.ts file of our Angular app.

This is the main App module. As you can see, it’s made up of four main parts:

* The declarations array
* The imports array
* The providers array
* The bootstrap array

📝 **Note:** There is another part of a module that is not shown in this example: the export array.

So, what do all these different arrays do? Well, let’s look at each one, as follows:

One of the first things to point out is the use of a class decorator to tell Angular about the details of this module. As you can see, the @NgModule is a decorator. Angular sees this and knows that this TypeScript class is a module and that the details within the @NgModule decorator are all parts of this module. So, through this decorator, Angular knows that this module has its own version of AppComponent that belongs to this module. It then imports another module called BrowserModule, and when Angular boots up, it should use the AppComponent as part of this Bootstrap process.

### declarations array

This contains the components, directives, and pipes that are part of this module.

### imports array

This contains other modules, whose classes are needed by components of the module they are being imported into.

### providers array

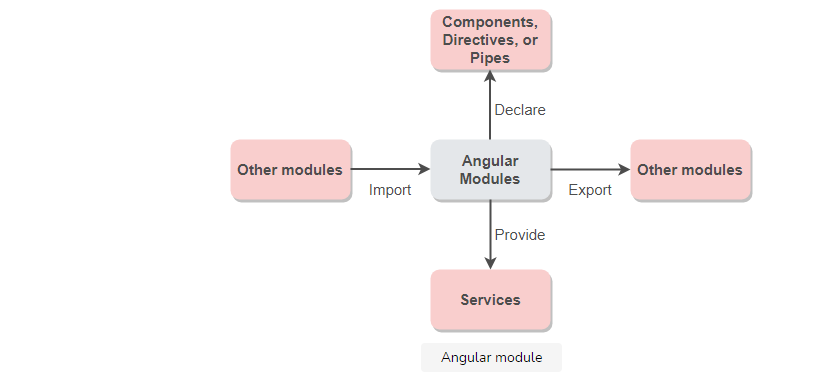
This contains any services that are required by components. If a service is added to the module level, it is available to all components that are part of the module, but services can also be imported at just the component level.

### bootstrap array

This contains the main component, or the root component, which starts the whole application. Only the root module (in our architecture application, it’s the app.module.ts file) that we have opened can have a bootstrap array.

### export array

This contains a list of declarations that are available by components in other modules.

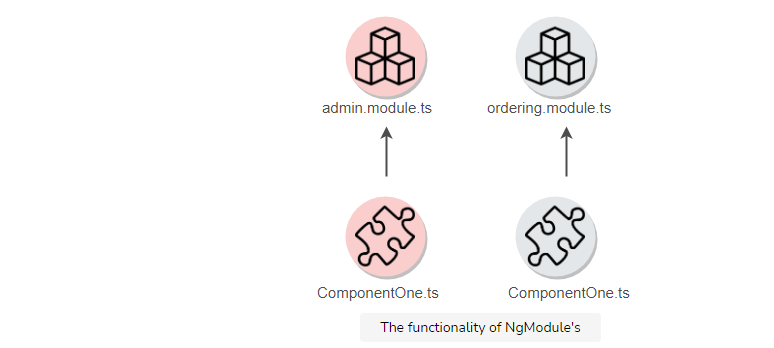


## The functionality of NgModules

NgModules main role is to tell the framework what components belong where when the application is being compiled. For example, suppose that I have a component called ComponentOne.ts, and in the same application, another developer working on the project also creates a new component and decides to call it ComponentOne.ts. The compiler wouldn’t know which ComponentOne to use when the application was running. By using a module, we can say that one ComponentOne belongs to this module, and the other one belongs to another module. Then, when the compiler is running the application, and it is running the code that belongs to a module, the compiler knows which ComponentOne file to use. This helps to group functionality together and allows a different developer to work on separate parts of an application without affecting the part of the application that another developer is working on.

With NgModule, we can say that one ComponentOne.ts belongs to the admin modules, admin.module.ts, and the other ComponentOne.ts belongs to the ordering module, ordering.module.ts. Each component has a context of where it belongs. So, Angular knows where each ComponentOne belongs and that they are separate components.

✏️ **Best coding practice:** Naming components the same name is never a good idea. It’s sometimes unavoidable, especially when incorporating a third-party library into your project.



📝 **Note:** We will be going further into NgModule in [NgModules chapter](https://www.educative.io/collection/page/10370001/4603693004488704/4518520179130368" \t "_blank), where we will not only look into a more complex module file but will also start to create modules for our demo app.

# Components

Let's dig deeper into the structure of the Angular component.

**We'll cover the following**

* [What are the components?](https://www.educative.io/courses/getting-started-with-angular/JPQ1vJzV3KK#What-are-the-components?)
* [Structure of component](https://www.educative.io/courses/getting-started-with-angular/JPQ1vJzV3KK#Structure-of-component)
  + [@Component decorator](https://www.educative.io/courses/getting-started-with-angular/JPQ1vJzV3KK#@Component-decorator)
  + [Property of a component class](https://www.educative.io/courses/getting-started-with-angular/JPQ1vJzV3KK#Property-of-a-component-class)

## What are the components?

**Components** are the building blocks of the application. They are single pieces of functionality in our application, which are linked together under a module.

Components can have visual elements to them, which allow the user to interact with the application.

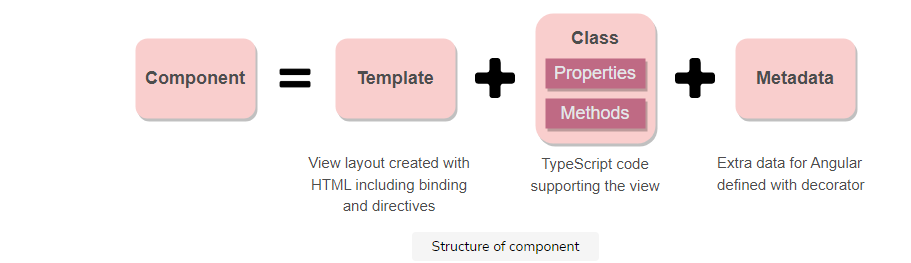
## Structure of component

Let’s go back to our angular-architecture demo application. If we open app.component.ts,

This is the entry component of our application. If you look at the app.module.ts file, you’ll see that AppComponent is set to be the Bootstrap component for the application. That means that this component will be the start of the application, and the template for this component will be the first thing a user will see when the app has loaded in the browser.

A component is made up of two main parts:

* The @Component decorator
* The component class.



### @Component decorator

Again, the component is a TypeScript class that is using the @Component decorator to tell Angular about the details of the component. In this @Component decorator, we can see that the component has an HTML template called app.component.html and a CSS file called app.component.css.

The @Component decorator also tells Angular that the selector, or HTML tag, for this component is app-root; this is the HTML that the selector generates:

<app-root></app-root>

The **selector name** is used to create the HTML tag that Angular knows about, so when that HTML tag is seen in other component templates, Angular knows what component to use and what component’s template to display.

### Property of a component class

In our example component within the class, we can see a property of the component class called title. This property is available in the associated template of the component, which you can see in the following section of app.component.html:

<div style="text-align:center">  
       <h1>Welcome to {{ title }}!</h1>  
</div>

There’s more that can be added to the component class besides properties that are available to the associated template/view. The functionality of the template is defined in the component class as well as data. Common functionality provided by Services is loaded into the component class, making it available to the component template.

We’ll be looking at components more closely in a [Components, Templates, and Forms in Angular chapter](https://www.educative.io/collection/page/10370001/4603693004488704/5860980926971904). We’ll be creating new components for our demo application and looking more closely at the relationship between the component class and the component template. But for now, this should give you an understanding of the basic structure of an Angular component.

# Services

Let’s take a look at Angular services: how they are structured, and how decorators are used for defining services in Angular.

**We'll cover the following**

* [What are the services?](https://www.educative.io/courses/getting-started-with-angular/N8my9qZZZmK#What-are-the-services?)
  + [Create modularity and reusability](https://www.educative.io/courses/getting-started-with-angular/N8my9qZZZmK#Create-modularity-and-reusability)
  + [Dependency Injection](https://www.educative.io/courses/getting-started-with-angular/N8my9qZZZmK#Dependency-Injection)
* [Structure of service](https://www.educative.io/courses/getting-started-with-angular/N8my9qZZZmK#Structure-of-service)
* [Create service for angular-architecture app](https://www.educative.io/courses/getting-started-with-angular/N8my9qZZZmK#Create-service-for-angular-architecture-app)
  + [Step 1: Create service using CLI command](https://www.educative.io/courses/getting-started-with-angular/N8my9qZZZmK#Step-1:-Create-service-using-CLI-command)
  + [Step 2: Add service to app.module.ts file](https://www.educative.io/courses/getting-started-with-angular/N8my9qZZZmK#Step-2:-Add-service-to-app.module.ts-file)
  + [Step 3: Add service to app.component.ts file](https://www.educative.io/courses/getting-started-with-angular/N8my9qZZZmK#Step-3:-Add-service-to-app.component.ts-file)

## What are the services?

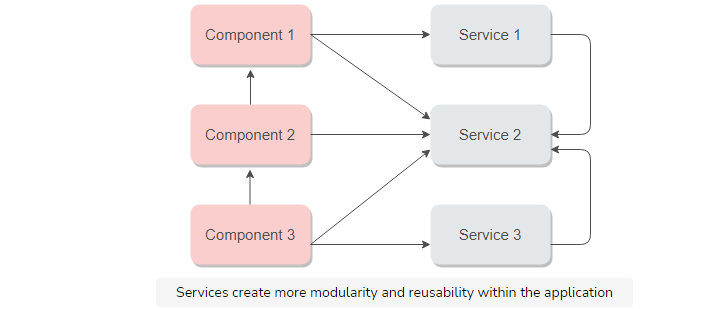
**Services** are single TypeScript classes used to access information and share it between components.

A **service** is simply a TypeScript class, similar to how a component is a TypeScript class.

### Create modularity and reusability

The main difference between a component class and a service is that services are used to create more modularity and reusability within the application. They are used as a way to **share functionality that may be needed more than once throughout the application**. This helps to improve the modularity of the application by dividing it into reusable and standalone services that components can call in. Each service performs a single piece of the app’s functionality. This leads to the **app being divided up into these smaller services**, rather than having all the logic of the app in one large, monolithic service or component.

Services are there to do only one thing well. This means that the component can load in services to do the one piece of logic the component requires; then, the component can call another service to perform another piece of logic that the component needs. This leads to a **component needing to have access to multiple services**. Angular loads services into components through Dependency Injection.

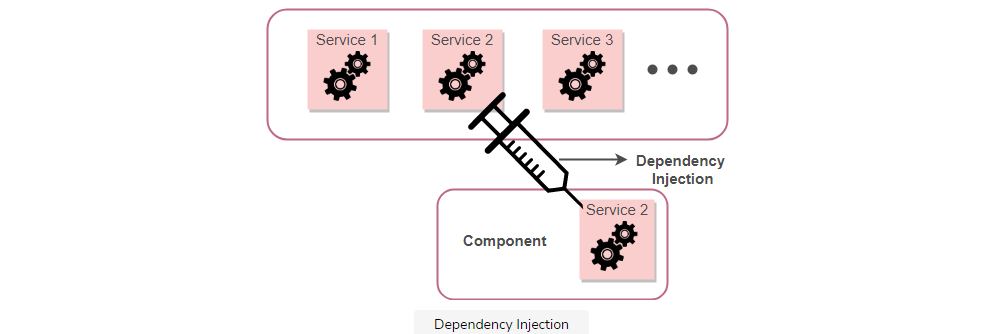


### Dependency Injection

**Dependency Injection (DI)** is the method that Angular uses to tell components what services they can consume.

DI is not just an Angular-specific concept; there are many frameworks that use dependency injection, and not just frontend frameworks.

Angular has always used DI. Even from the early versions of AngularJS, DI has been the method that Angular has used to inject services into components.



In our angular-architecture project, we don’t have a service automatically generated for us by the Angular CLI. This is because (as we know), services are used to manage data and logic within our application. This changes from application to application. The Angular CLI team couldn’t get the CLI to generate a service for us that fits the needs of our application. It’s impossible. So, they don’t provide a service for a very basic Angular application. We have to create one ourselves.

## Structure of service

While the angular-architecture application doesn’t have a service, we can still take a look at an example to see the structure of a service.

In the Angular official documentation, there is an example application called [Tour of Heroes](https://angular.io/tutorial), and it is possible to download and view the source code of this example application.

📝 **Note:** It’s well worth doing this, as the **Tour of Heroes** application has great examples of the various parts of an Angular application. It was written by some of the leading experts within the Angular community, so it’s a great example of some best practices for building Angular apps.

In this **Tour of Heroes** application, there are many services that we can take a look at to see how a service is structured. This is one of the main services, which loads a list of Heros from an external API:

export class HeroService {  
    private heroes: Hero[] = [];  
    constructor(private backend: BackendService, private logger: Logger) {}  
  
    getHeroes() {  
        this.backend.getAll(Hero).then((heroes: Hero[]) => {  
            this.logger.log(`Fetched ${heroes.length} heroes.`);  
            this.heroes.push(...heroes); // fill cache  
        });  
        return this.heroes;  
    }  
}

As you can see, it is a simple TypeScript class with a constructor, and a single method called getHeros(). It also calls another service, BackendService. This shows an example of the modularity that services provide, where one service does one single task and uses another to perform another task, in this case providing data.

What’s missing from this service is the decorator that tells Angular that this class can be consumed/injected into components. The decorator that provides Angular with this information is called the Injectable decorator, and it looks like this:

@Injectable({  
    providedIn: "root",  
})  
export class HeroService {}

When the @Injectable decorator is added to a class, this tells Angular that this class and its functionality can be passed into components and other services, using Dependency Injection.

## Create service for angular-architecture app

To get a better understanding of how a service is linked to a module, let’s create a simple Service for our angular-architecture app.

### Step 1: Create service using CLI command

We can create a service by running the following command in CLI:

ng generate service demo

📝 **Note:** Click the terminal window and see what happens!

This will tell the Angular CLI to create a new service called demo within our source code. When we execute this command, it will create demo.service.spec.ts and demo.service.ts files within the app folder.

So, you should have a nice new service called demo.service.ts, which looks like the following:

import { Injectable } from '@angular/core';  
  
@Injectable({  
  providedIn: 'root'  
})  
export class DemoService {  
  
  constructor() { }  
}

Pretty impressive? Well, not really, because it doesn’t actually do anything yet, but it does demonstrate how the TypeScript class is using the @Injectable decorator so that Angular knows that this class can be injected into components and other services via DI.

### Step 2: Add service to app.module.ts file

Now that we have this service let’s add it to our app.module.ts file so that Angular knows what context this service belongs under.

Open the app.module.ts file, which should look like this:

import { BrowserModule } from "@angular/platform-browser";  
import { NgModule } from "@angular/core";  
import { AppComponent } from "./app.component";  
@NgModule({  
    declarations: [AppComponent],  
    imports: [BrowserModule],  
    providers: [],  
    bootstrap: [AppComponent],  
})  
export class AppModule {}

To add the service, we type the name of the service to the providers array, like this:

providers: [  
    DemoService  
],

Now, set the import statement at the top to where the service TypeScript class is stored, as shown here:

import { AppComponent } from './app.component';  
import { DemoService } from './demo.service';

### Step 3: Add service to app.component.ts file

The final stage is to have the app.component.ts consume the new DemoService. To do that, we add the service to the constructor of the component class, as follows:

import { Component } from '@angular/core';  
import { DemoService } from './demo.service';  
  
@Component({  
  selector: 'app-root',  
  templateUrl: './app.component.html',  
  styleUrls: ['./app.component.css']  
})  
export class AppComponent {  
  title = 'angular-architecture';  
  constructor(public demoService: DemoService) {}  
}

In **line 11** of the app.component.ts file, we have given the DemoService a local name of demoService, so if we want to access any of the functions of the service within our component class, we use the following local reference:

this.demoService.printHello();

Here is our app source code with the demo service and the updated app.module.ts and app.component.ts files.

📝 **Note:** Press the **RUN** button to compile and serve the application. Once the app compiles, you can see its output in the **Output** tab or by clicking on the URL given after Your app can be found at.



# Why Use TypeScript?

Learn why Angular is built on TypeScript and what benefits this brings to Angular developers.

**We'll cover the following**

* [What is TypeScript?](https://www.educative.io/courses/getting-started-with-angular/7n9G7E1XV91#What-is-TypeScript?)
* [Why is Angular written in TypeScript?](https://www.educative.io/courses/getting-started-with-angular/7n9G7E1XV91#Why-is-Angular-written-in-TypeScript?)
  + [Introduction of a new feature is slow in JavaScript](https://www.educative.io/courses/getting-started-with-angular/7n9G7E1XV91#Introduction-of-a-new-feature-is-slow-in-JavaScript)
  + [Powerful tooling](https://www.educative.io/courses/getting-started-with-angular/7n9G7E1XV91#Powerful-tooling)
  + [Make abstractions using interfaces](https://www.educative.io/courses/getting-started-with-angular/7n9G7E1XV91#Make-abstractions-using-interfaces)
  + [More readable and easy to write code](https://www.educative.io/courses/getting-started-with-angular/7n9G7E1XV91#More-readable-and-easy-to-write-code)

## What is TypeScript?

In AngularJS, everything was written in JavaScript. With the new version of Angular, the decision was made to write everything using Typescript. There are many reasons for this change, but first, let’s take a look at what TypeScript is.

[TypeScript](https://www.typescriptlang.org/) is from Microsoft, and according to the official website:

What this means is that TypeScript has a larger set of features that JavaScript doesn’t have, but when you save your TypeScript file, the TypeScript compiler transforms the **TypeScript code to plain JavaScript**. Let’s see why Angular needs TypeScript?

## Why is Angular written in TypeScript?

Let’s see why Angular is written in TypeScript and what benefits it brings for Angular developers.

### Introduction of a new feature is slow in JavaScript

The simple reason is that JavaScript, while a fantastic language, moves slowly when adding new features. For a new feature to be added to the language, it has to be discussed by all the relevant committees (JavaScript, while not being owned by anyone, is governed by a committee of community leaders and representatives for the various web browser developers). These committees need to discuss a new feature:

* How will it be implemented?
* How will it be used by developers?
* What benefits does it bring to our application?

Then, they make it part of the ***ECMAScript standard***, which is a document that sets out what languages based on ECMAScript should be able to do.

Then, all the various browsers need to start adding support for these new features of JavaScript into their browser, and the JavaScript engine, which is part of the browser. Therefore, when one browser supports a new feature, this allows a developer to use that feature and for it to work in the browser. Another browser company may not have implemented it, so when a user views a website or web application using this new feature, it won’t work for them.

The introduction of new features is a slow process in JavaScript. It has become faster over the last few years as the popularity of JavaScript has grown, but there are still a lot of hurdles for a new language feature to be added to plain JavaScript.

Microsoft decided that they would create a new language—not to replace JavaScript (like they tried with JScript a few years ago), but one that would have the features they believed a modern web language should have. With their knowledge and experience in developing tools, they would create a compiler for this new language that would generate JavaScript that used its set of features to replicate the new features of this new language. This would allow the developer to use the latest features of a modern web language and leave the compiler to work out how to mimic these features in JavaScript.

TypeScript compilation process

This is what TypeScript aims to do; Microsoft can add all the new features they want to TypeScript: features they believe a language should have based on their experience with other languages, such as C#. These features include types, interfaces, and modules (although these are now available in JavaScript), and there is no wait for these features to be available to developers. Developers know that support for these new language features is made by the TypeScript compiler.

### Powerful tooling

Angular is written in TypeScript because it provides so much insight and powerful tooling. One of the core developers of Angular, **Victor Savkin**, said the following:

“The biggest selling point of TypeScript is tooling. It provides advanced autocompletion, navigation, and refactoring. Having such tools is almost a requirement for large projects. Without them, the fear of changing the code puts the codebase in a semi-read-only state and makes large-scale refactorings very risky and costly.”

📝 **Note:** You can find this article by Victor Savkin [here](https://vsavkin.com/writing-angular-2-in-typescript-1fa77c78d8e8?gi=aedc8398425a).

Not only does TypeScript provide great features that aren’t available in JavaScript, but it also provides great tooling for developers to use in order to find issues and bugs within their code before it’s even run in the browser.

For example, as we go through building out these applications, TypeScript, and more specifically, the TypeScript compiler, is inspecting our code and is aware of all the different parts of our code. If we add a property to a class then use that class in another class (as we did when we added our demo service to our main component class), the TypeScript compiler knows what functions are available from this demo service. Therefore, when we go to use the service in the component class, the code editor knows what is available and gives us insight into what functions are available as we type.

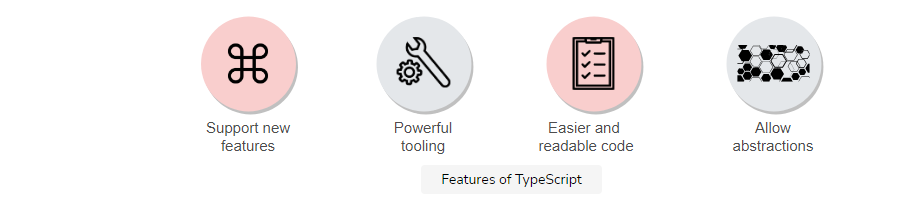
The main benefit that TypeScript provides is this insight into our code as we write. An application can become more and more complex, and trying to remember all the moving parts is extremely hard, but with TypeScript and the great tooling it gives us, there is **less** we need to remember and a lower chance of bugs appearing in the code.

### Make abstractions using interfaces

As well as great tooling, TypeScript allow us to make abstractions when we are defining the model of our application. Using **interfaces**, which TypeScript provides us, allows us to define how our application will be structured and the relationships between the parts of the data model that our application uses. This is something we will be looking at in the [next lesson](https://www.educative.io/collection/page/10370001/4603693004488704/4822181581160448), where we’ll go over the architecture of our Client Contacts application.

### More readable and easy to write code

Moving from JavaScript to TypeScript can be difficult at the start. When you start writing TypeScript, the compiler keeps telling you the mistakes and problems with your code as you type, while JavaScript would just let you get away with these simple mistakes. Therefore, you would think that JavaScript is quicker to develop with, but the more and more you work with TypeScript, the better your code becomes, and the better you become. Soon, you’ll be writing code that is easier to read and understand.



# The Architecture of Our Client Contacts Manager Application

Let's take a look at the high-level architecture of our Client Contacts Manager application.

**We'll cover the following**

* [Application features](https://www.educative.io/courses/getting-started-with-angular/myvMkoqBR93#Application-features)
* [Application model](https://www.educative.io/courses/getting-started-with-angular/myvMkoqBR93#Application-model)
  + [Salesperson interface](https://www.educative.io/courses/getting-started-with-angular/myvMkoqBR93#Salesperson-interface)
  + [Contact interface](https://www.educative.io/courses/getting-started-with-angular/myvMkoqBR93#Contact-interface)
  + [Company interface](https://www.educative.io/courses/getting-started-with-angular/myvMkoqBR93#Company-interface)
* [Components and services](https://www.educative.io/courses/getting-started-with-angular/myvMkoqBR93#Components-and-services)

## Application features

We’ve already spoken about it a couple of times, but throughout this course, we’ll be building a **Client Contacts Manager** application for the desktop browsers using Angular. Since we’ve been looking at the Angular architecture, we are now going to take a high-level look at the architecture of the Angular version.

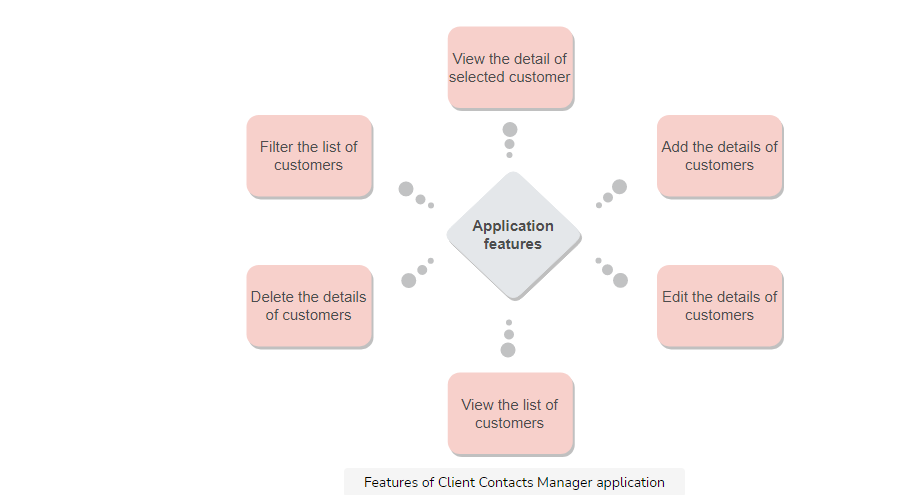
The main features of this application are as follows:

* View a list of contacts
* Search through the available contacts
* View the details of a selected contact
* Edit the details of a contact
* Add a new contact to the system
* Remove a contact

There’s also going to be a Company section because each contact must belong to a company, as our fictional salesperson needs to be able to find the contacts for a particular company.

So, in the Company section, the salesperson will be able to do the following:

* View all the companies
* Search for a company by the name
* View the details of a selected company
* Edit the details of a selected company
* Add a new company to the system
* Remove a company from the system



## Application model

If we think about the model behind this application, there are three main data models:

* the contact
* the company
* the salesperson who uses the application.

Each of them can have an Interface that defines the details of the model and what each model representation can do.

### Salesperson interface

So, the interface for the salesperson could be as follows:

interface SalesPerson {  
       firstname: string;  
       lastname: string;  
       email: string;  
       password: string;  
       jobtitle: string;  
       startDate: Date;  
       active: boolean;  
       accessLevel: number;   
       relatedCompanies: Companies[];  
       relatedContacts: Contacts[];  
   }

Here, we have set some properties that a salesperson will have: firstname, email, startDate, and so on. We’ve also created properties that they may have for relatedCompanies and relatedContacts. Both of these are defined as arrays with a type of Companies and Contacts, which are types we define for the data model of the other sections of the application.

### Contact interface

The Contact interface looks as follows:

interface Contact {  
       id: number;  
       firstname: string;  
       lastname: string;  
       email: string;  
       jobtitle: string;  
       status: string;  
       isActive: boolean;  
       relatedCompany: Company;  
       notes: string;  
}

Here, we are defining the model of a contact to have an id; a firstname and last name; some details about their job, such as the status; and the Company they belong to, which is set as the relatedCompany property that is a type of Company.

### Company interface

Finally, our Company interface will look as follows:

interface Company {  
    id: number;  
    name: string;  
    address1: string;  
    address2: string;  
    town: string;  
    city: string;  
    postCode: string;  
    country: string;  
    contactEmail: string;  
    numberOfStaff: string;  
    industry: string;  
    isActive: boolean;  
}

As you can see, we’ve created a Company type that has a name, address details, some information about the numberOfStaff, and the industry the company works in. This Company type is the type that is used for the relatedCompany property of the Contact type.

Now that we have a few interfaces mapped out, it shows how the features of TypeScript allow us to abstract the model of the application into types that our application will use.

## Components and services

What else do we need for our application? Well, here’s an initial list of the components and services we need for this application:

* Main app component
* Main nav component
* Header component
* Login form component
* Contact form component
* Company form component
* Add a new contact button component
* Search form component
* List of contacts component
* Service to manage the salesperson’s access
* Service to manage the contacts
* Service to manage the company
* Service to manage saving data to an external data source
* All the CSS for styling the app

That’s a lot of components and services, and there may be more. We won’t create these now. Instead, we’ll start to create these components and services using the Angular CLI.

# Creating and Running the Client Contacts Manager Application

Let's create and run our project application using the Angular CLI.

**We'll cover the following**

* [Step 1: Navigate to the working directory](https://www.educative.io/courses/getting-started-with-angular/NEZpMVwkBnp#Step-1:-Navigate-to-the-working-directory)
* [Step 2: Create a new Angular application](https://www.educative.io/courses/getting-started-with-angular/NEZpMVwkBnp#Step-2:-Create-a-new-Angular-application)
* [Step 3: Change the current working directory](https://www.educative.io/courses/getting-started-with-angular/NEZpMVwkBnp#Step-3:-Change-the-current-working-directory)
* [Step 4: Run Angular application](https://www.educative.io/courses/getting-started-with-angular/NEZpMVwkBnp#Step-4:-Run-Angular-application)
* [Expected output](https://www.educative.io/courses/getting-started-with-angular/NEZpMVwkBnp#Expected-output)
* [Use ng serve -o to improve the development process](https://www.educative.io/courses/getting-started-with-angular/NEZpMVwkBnp#Use-ng-serve--o-to-improve-the-development-process)

Let’s create the **Client Contacts Manager** application.

## Step 1: Navigate to the working directory

First, let’s go to our main working folder. We like to keep our projects in an educative folder. Let’s navigate to the working folder using the following command:

cd usr/local/educative

## Step 2: Create a new Angular application

Once we’re there, we need to tell the Angular CLI to create a new Angular application. To tell the Angular CLI to create an application, we simply use the following command:

ng new Client-Contacts-Manager-Angular

This will create a new folder within the educative folder called Client-Contacts-Manager-Angular, where the CLI will create all the shellcode of the application. As the CLI runs, it will ask you the following two questions:

1. Would you like to add Angular routing?

2. Which stylesheet format would you like to use?

Basically, the CLI is asking: Do we want to add routing to the application? ***Routing*** is how you set up navigation throughout an application, and by saying yes, we are asking the CLI to create a Routing module, where all our links can be added to a separate module. We will be going over Routing in the [Routing and Navigation chapter](https://www.educative.io/collection/page/10370001/4603693004488704/5920025016795136), but for now, say Yes.

The second question asks: What CSS format do we want to use in this application? We can use plain CSS for the application, but we are going to choose SCSS.

## Step 3: Change the current working directory

After agreeing to these two questions, the CLI will create a folder called Client-Contacts-Manager-Angular and add all the scaffolding code for the application. Once the CLI has finished, cd into the folder that has just been created:

cd Client-Contacts-Manager-Angular

## Step 4: Run Angular application

Now that we have our application generated by the CLI, it’s a perfect time to run the application and see what the CLI has created for us. Not only will the CLI build an application for us, but it will run the application in the browser itself. To do this, we need to type the following command within the terminal:

ng serve

Or alternatively, you can run this command:

npm run

📝 **Note:** We have done all the initial setup for you. Press the **RUN** button to compile and serve the application. Once the app compiles, you can see its output in the **Output** tab or by clicking on the URL given after Your app can be found at.

What we have here is our application running in the browser. As part of building a new application, the CLI generates an initial page with some helpful links. These links are as follows:

* [Learn Angular](https://angular.io/tutorial): A demo application created by the Angular team, which you can read about in the Angular documentation and see a live demo.
* [CLI Documentation](https://angular.io/cli): A link to the wiki page of the Angular CLI team with details on how to use the CLI and a list of other commands.
* [Angular blog](https://blog.angular.io/): The Medium blog of the Angular team, well worth reading and regularly checking to see what the team is up to.

## Use ng serve -o to improve the development process

If you run the ng serve command on your local machine, you have to open your favorite browser and then navigate to this URL to load the application:

http://localhost:4200

Now, as we’ve seen, running the app and then opening the browser is easy, but there is an even easier way to get the application running in the browser. We can tell the CLI to not only start the application but open the browser and load the application for us.

So if you are working on your local machine, close the browser and in the terminal use CMD/CTRL + C to stop the Node service running. Then, within the same Terminal window, type the following command:

ng serve -o

This tells the CLI to compile the application, start the Node service running again, and open the browser loading the application again at http://localhost:4200.

You should now see the same web application running in your browser as we did before, but this time there was no need to go and open the browser and enter the URL of the site. The Angular CLI has done all this for you.

This is a small example of how Angular can help improve the development processes. There are many commands that the CLI provides, which you can use to tell the CLI to help you as you build out your Angular application.

# Commands of the Angular CLI

Let's look at some of the Angular commands and their purpose.

**We'll cover the following**

* [What is a command?](https://www.educative.io/courses/getting-started-with-angular/my4kGpwBR6A#What-is-a-command?)
* [Angular commands](https://www.educative.io/courses/getting-started-with-angular/my4kGpwBR6A#Angular-commands)
* [Practice time](https://www.educative.io/courses/getting-started-with-angular/my4kGpwBR6A#Practice-time)

## What is a command?

So we know the Angular CLI can be used to start a new application and also run the new application in the browser, but that’s not all it can do. There are other commands which can be used as we are developing our applications.

A **command** is a term for instructions we type into the Command Line Interface (CLI).

## Angular commands

Let’s see a list of available commands:

| **Angular Command** | **Purpose** |
| --- | --- |
| **ng add** | Used to add third-party libraries to an existing application, we will be using this soon to add Angular Material |
| **ng build** | Used to compile the complete application into a /dist folder or a folder provided by an argument |
| **ng config** | Allows you to either view or set configuration settings for your app; these configuration settings can be passed as JSON |
| **ng doc** | Opens the official Angular docs website; if you want to find a specific topic, add the keyword as an argument |
| **ng e2e** | This will run the end-to-end tests of the application |
| **ng generate** | Command to create/generate new components, services, and other parts of your Angular application. We will use this command a lot over the next few chapters |
| **ng help** | Provides a help menu for the Angular CLI |
| **ng lint** | Runs linting over your application’s codebase |
| **ng new** | Starts the process to create a new Angular application application |
| **ng run** | Starts running a custom target for your application. In your package.json file, you can add custom commands that ng run will start for you |
| **ng serve** | Starts the local Node server so you can access the site in the browser |
| **ng test** | Starts the running of all the Unit Tests you create for your application |
| **ng update** | Updates the application and any dependencies in the application, very useful when a new version of Angular is released |
| **ng version** | Tells you the version number of the Angular CLI currently being used |
| **ng xi18n** | Extracts any xi18n messages within your application, used as part of adding multi-language support to your application |

That is a lot of commands, and there may be more coming in future releases of the Angular CLI. This shows how much you can do with the CLI and how you will rely on it as part of your day-to-day Angular development.

## Practice time

You can execute the above-mentioned commands within the terminal window.

To do this, we need to open up the terminal and navigate to the newly created angular folder using the following command:

cd usr/local/educative/angular

Then we can execute any of the above commands in the angular project.

📝 **Note:** Click the terminal window and see what happens!

# Run Unit Tests Using Angular CLI

Let's see how we can use the Angular CLI to run unit tests.

**We'll cover the following**

* [Testing in Angular](https://www.educative.io/courses/getting-started-with-angular/RM0pNgWpxl0#Testing-in-Angular)
* [Run unit tests](https://www.educative.io/courses/getting-started-with-angular/RM0pNgWpxl0#Run-unit-tests)
* [Expected output](https://www.educative.io/courses/getting-started-with-angular/RM0pNgWpxl0#Expected-output)

## Testing in Angular

Unit Tests are an important part of modern web application development. Angular creates tests for every component or service automatically when you use the ng generate command (though you can pass an argument to the generate command to not create a test). If you look at the codebase of our Angular application, you’ll see many examples of these files ending in spec.ts. These are the test files that the Angular CLI has created for us.

Testing in Angular

## Run unit tests

We will be going further into Unit Testing in a later chapter, but what we really want to look at is how we can use the CLI to run these and our future tests for us.

Again, open Terminal and navigate to our Angular folder. Once there, enter the following command:

ng test

Clicking the **Run** button will automatically execute the above-mentioned commands.

# Components, Templates, and Forms in Angular

Let's learn about Components, Templates, and Forms in Angular..

**We'll cover the following**

* [Chapter learning outcomes](https://www.educative.io/courses/getting-started-with-angular/m2PEl3BQVYE#Chapter-learning-outcomes)
* [Client contacts manager application](https://www.educative.io/courses/getting-started-with-angular/m2PEl3BQVYE#Client-contacts-manager-application)

## Chapter learning outcomes

We’ve covered a lot over the last few chapters. We’ve looked at the **architecture** of an Angular application and we’ve gone through the **Angular CLI**. Now, we are going to start looking at the building blocks of an Angular application – that is, its **components**.

In this chapter, you will learn about the following topics:

* What are components and how are they structured?
* What are smart and dumb components?
* How to pass data between components.
* What different types of templates are supported by Angular.
* Why you should categorize your components.
* How to add support for forms to the application.

We’ve already seen what a component looks like in the [Angular Architecture chapter](https://www.educative.io/collection/page/10370001/4603693004488704/5049007880536064), but we’ve only looked at one component: one that doesn’t have a great deal of functionality. In this chapter, we are going to be delving further into components. We are going to look at the **structure of a component**, the **supporting component classes**, as well as **component templates**. Then, we are going to look at how we can use components within our Angular application, how to pass data between components, and how to use components to structure our application.

Once we’ve been through what components are, we will look at **templates** in Angular, how they are linked to components, and the different ways templates can be defined, as well as how events within a template can be set up, allowing you to add interactivity to your templates.

Then, we will go through **forms** in Angular, looking at the various types of forms that Angular supports and how you can add forms to your application.

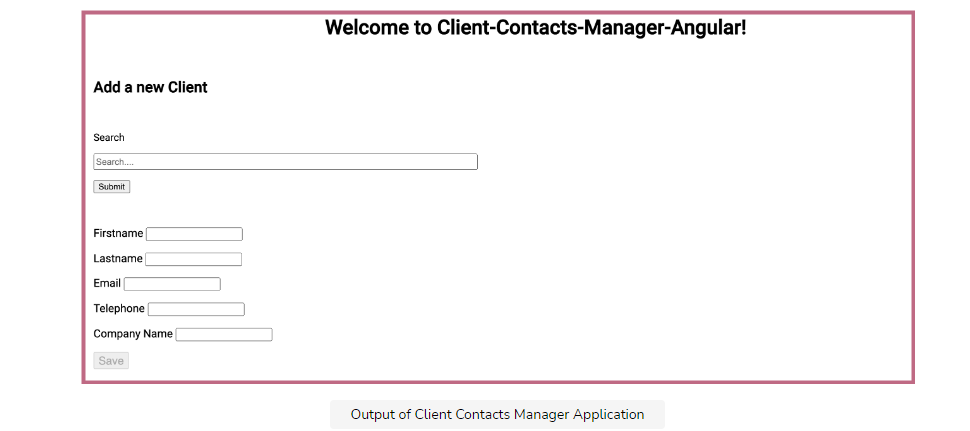
## Client contacts manager application

Now, we will start adding components and templates to our client contacts manager application in order to start building out the application.

📝 **Note:** Press the **RUN** button to compile and serve the application. Once the app compiles, you can see its output in the **Output** tab or by clicking on the URL given after Your app can be found at.



You should see the following in the output:



# The Structure of a Component

Let's dissect a component into its main parts and discuss each of them in detail.

**We'll cover the following**

* [@Component decorator](https://www.educative.io/courses/getting-started-with-angular/x1MYvp08Y1E#@Component-decorator)
  + [selector](https://www.educative.io/courses/getting-started-with-angular/x1MYvp08Y1E#selector)
  + [templateUrl](https://www.educative.io/courses/getting-started-with-angular/x1MYvp08Y1E#templateUrl)
  + [styleUrls](https://www.educative.io/courses/getting-started-with-angular/x1MYvp08Y1E#styleUrls)
* [The component class](https://www.educative.io/courses/getting-started-with-angular/x1MYvp08Y1E#The-component-class)
  + [Properties](https://www.educative.io/courses/getting-started-with-angular/x1MYvp08Y1E#Properties)
  + [Methods](https://www.educative.io/courses/getting-started-with-angular/x1MYvp08Y1E#Methods)
  + [MVC pattern and Component class](https://www.educative.io/courses/getting-started-with-angular/x1MYvp08Y1E#MVC-pattern-and-Component-class)

A component is made up of two main parts:

* The @Component decorator
* The component class.

## @Component decorator

The following example shows a component. As you can see, there is the @Component decorator, which has three parts to it:

* The selector
* The template URL
* The styles URL
* For example, here is a Component class:
* import { Component } from "@angular/core";  
  @Component({  
      selector: "app-root",  
      templateUrl: "./app.component.html",  
      styleUrls: ["./app.component.scss"],  
  })  
  export class AppComponent {  
      title = "Welcome to your component";  
      yourName: string;  
      showForm = false;  
      constructor() {}  
      displayYourName(name: string) {  
          this.yourName = name;  
      }  
      toggleDisplay() {  
          this.showForm = true;  
      }  
  }
* In this Component class, we have a few things going on, in the @Component decorator (a Component class uses the @Component decorator to tell Angular what it is).

### selector

* The selector is the name that’s used to create the tag name of the component. For example, in this component, we have a selector called app-root that will generate the following tag:
* <app-root></app-root>
* Let’s change the selector to the following:
* selector: 'my-app-root'
* The HTML tag it would generate would be as follows:
* <my-app-root></my-app-root>
* You can actually set the tag prefix for your application.

### templateUrl

* We set what template this Component class uses by defining a URL to the template HTML file.
* templateUrl: "./app.component.html"

### styleUrls

* We then define the CSS file(s) that defines the style for this component.
* styleUrls: ["./app.component.css"]

## The component class

* Below the @Component decorator, we have the TypeScript class of our component.
* export class AppComponent {  
      title = "Welcome to your component";  
      yourName: string;  
      showForm = false;  
      constructor() {}  
      displayYourName(name: string) {  
          this.yourName = name;  
      }  
      toggleDisplay() {  
          this.showForm = true;  
      }  
  }

### Properties

* Here, we have a few properties (title, yourName, and showForm). These properties are available for use in the HTML template we linked in the @Component decorator.

### Methods

* Then, we have some methods (displayYourName and toggleDisplay), which are also available in the template. For example, the toggleDisplay method could be attached to a click event of a button in our template. So, every time the button is clicked, the toggleDisplay method is run, and the showForm property is set to true. This is not a great example of a method, but it’s here to show how the Component class is used to define the business logic of the component.

# The Component Life Cycle Hooks

Let's explore how component life cycle hooks can run logic as the component is being created and is displayed in the UI.

**We'll cover the following**

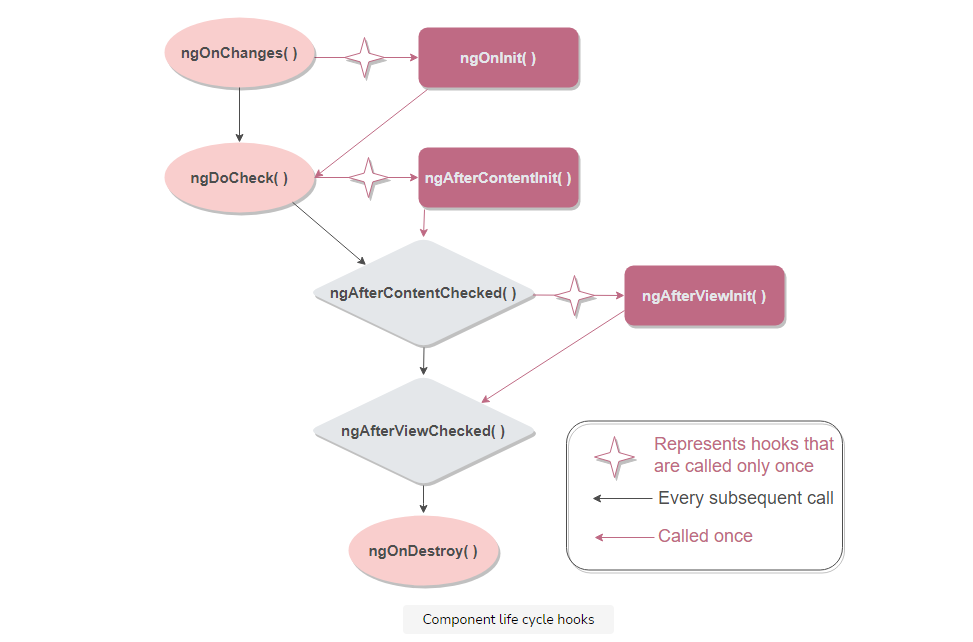
* [The life cycle of Angular components](https://www.educative.io/courses/getting-started-with-angular/R8KvlvE5JxE#The-life-cycle-of-Angular-components)
* [Life cycle hooks](https://www.educative.io/courses/getting-started-with-angular/R8KvlvE5JxE#Life-cycle-hooks)
* [Running custom code in ngOnInit() hook](https://www.educative.io/courses/getting-started-with-angular/R8KvlvE5JxE#Running-custom-code-in-ngOnInit()-hook)

## The life cycle of Angular components

Angular manages all the components we make for our applications in the same way it manages all of its components, whether they are ones we’ve created or third-party components, such as Angular Material. All components go through the same life cycle, where Angular creates the component, renders it, then creates and renders any child components. Angular also handle changes in data properties within our component and when a component is removed from the view, for example, when navigating to a new view.

## Life cycle hooks

All of these stages have life cycle hooks that we can tap into and add our own logic to so that we can have our application do something while a component is going through one of these stages. For example, we may want to load some data before the component is rendered to the view. We could do that as part of the OnInit life cycle hook. But if we want to make sure that the latest data is loaded, we might want to use the OnChange life cycle hook, which is run many more times than the OnInit life cycle hook.



Here is a table of the available component life cycle hooks:

| **Hook** | **Purpose** |
| --- | --- |
| **ngOnChanges()** | Responds when Angular either sets a data property or when an existing data property is reset. For example, a dynamic label on a page. |
| **ngOnInit()** | Initializes the component after Angular has set all the data properties. This hook is only called once while ngOnChanges is called many times throughout a component’s life. |
| **ngDoCheck()** | This hook can be called to detect and act upon any changes we need Angular to know about that Angular can’t actually deal with. We can use this if we know there is an event that we want Angular to act upon, but Angular doesn’t know about it. For example, ngOnChange doesn’t pick up on an event. We can use ngDoCheck as we know it will be run after every ngOnInit and ngOnChange. |
| **ngAfterContentInit()** | This hook is available after Angular has added content to a component. This is useful any time we want to do something as soon as any content has been injected into a component from an external source. We could translate text in this hook as an example of what we could do in this step. |
| **ngAfterContentChecked()** | This hook is available once content has been added to a component. With this hook, we can check whether the content that’s been loaded into a component is correct or contains something we require. |
| **ngAfterViewInit()** | This is available after Angular has initialized the view of a component. We could update the model if a certain component UI has successfully loaded, maybe as part of the application’s security. |
| **ngAfterViewChecked()** | This hook is available after the component’s view has been checked, and the component’s child components views have been checked. We could use this to run a check to make sure a complex UI component’s children components have loaded properly. |
| **ngOnDestroy()** | This hook is available to us when a component is about to be removed from the UI. Here, we usually perform a lot of cleanup code, unsubscribe from Observables, and clean up event handlers. All of this helps improve application performance. |

## Running custom code in ngOnInit() hook

When we say that we can hook into these life cycle hooks/events, we mean that we can add our own code for ngOnInit() because we know Angular will run ngOnInit for a component. By creating our own ngOnInit() event handler in our Component class, we know that Angular will also call our business logic as part of the component’s life cycle.

For example, in a Component class, we could do the following:

export class AppComponent implements OnInit {  
  title = "Client-Contacts-Manager-Angular";  
  ngOnInit() {  
    console.log("My component is alive");  
}

Here, we have hooked into the ngOnInit() life cycle hook of this component and added a simple console.log statement, which we will see when the component is displayed in the view.

📝 **Note:** Press the **RUN** button to compile and serve the application. Once the app compiles, you can see its output in the **Output** tab or by clicking on the URL given after Your app can be found at.

# Interpolation and One-way Data Binding

Let's see how we can use interpolation for displaying data in our component's template.

**We'll cover the following**

* [What is interpolation?](https://www.educative.io/courses/getting-started-with-angular/NE65v95znzv#What-is-interpolation?)
* [What is one-way data binding?](https://www.educative.io/courses/getting-started-with-angular/NE65v95znzv#What-is-one-way-data-binding?)
* [String interpolation in action](https://www.educative.io/courses/getting-started-with-angular/NE65v95znzv#String-interpolation-in-action)
* [Return read values within the brackets](https://www.educative.io/courses/getting-started-with-angular/NE65v95znzv#Return-read-values-within-the-brackets)
  + [Exercise](https://www.educative.io/courses/getting-started-with-angular/NE65v95znzv#Exercise)
  + [Solution](https://www.educative.io/courses/getting-started-with-angular/NE65v95znzv#Solution)

## What is interpolation?

We know that we have this set of life cycle hooks we can extend upon with our own code, but how does Angular handle data displayed properties in our component’s template? This is all part of interpolation.

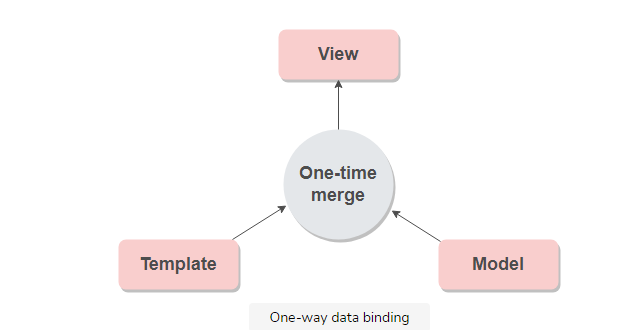
**Interpolation**, or string interpolation, is where you see the text within the **{{ }}** brackets, telling Angular that it should interpret the text within these brackets and use one-way data binding to replace the text within the **{{ }}** brackets with the value of the property of the same name.

So, what does that mean? Well, when we talk about interpolation, we need to look at one-way data binding.

## What is one-way data binding?

**One-way data binding** is where you have a property in your Component class, and when the value of that property is updated, this value is passed to the view.

The flow of data goes one way from the Component class into the view, which is why it’s called one-way data binding.



# Passing Data Between Parent/Child Components

Let’s look at ways in which parent and child components pass data between each other.

**We'll cover the following**

* [Passing data from the parent component to the child component](https://www.educative.io/courses/getting-started-with-angular/N7rJ2Yp2yl6#Passing-data-from-the-parent-component-to-the-child-component)
  + [Using @Input() decorator](https://www.educative.io/courses/getting-started-with-angular/N7rJ2Yp2yl6#Using-@Input()-decorator)
    - [Parent component](https://www.educative.io/courses/getting-started-with-angular/N7rJ2Yp2yl6#Parent-component)
    - [Child component](https://www.educative.io/courses/getting-started-with-angular/N7rJ2Yp2yl6#Child-component)
* [Passing data from the child component to the parent component](https://www.educative.io/courses/getting-started-with-angular/N7rJ2Yp2yl6#Passing-data-from-the-child-component-to-the-parent-component)
  + [@ViewChild() decorator](https://www.educative.io/courses/getting-started-with-angular/N7rJ2Yp2yl6#@ViewChild()-decorator)
    - [Parent component](https://www.educative.io/courses/getting-started-with-angular/N7rJ2Yp2yl6#Parent-component)
    - [Child component](https://www.educative.io/courses/getting-started-with-angular/N7rJ2Yp2yl6#Child-component)
  + [Using @Output decorator and EventEmitter class](https://www.educative.io/courses/getting-started-with-angular/N7rJ2Yp2yl6#Using-@Output-decorator-and-EventEmitter-class)
    - [Parent component](https://www.educative.io/courses/getting-started-with-angular/N7rJ2Yp2yl6#Parent-component)
    - [Child component](https://www.educative.io/courses/getting-started-with-angular/N7rJ2Yp2yl6#Child-component)
  + [Which approach is better?](https://www.educative.io/courses/getting-started-with-angular/N7rJ2Yp2yl6#Which-approach-is-better?)

There are different ways to pass data in and out of components. Choosing which approach to take depends on the complexity of the application you are developing and the relationship between the components.

The first approach we’re going to look at is the parent/child relationship, where one component has a child component.

## Passing data from the parent component to the child component

### Using @Input() decorator

The flow of data in this instance is from the parent to the child. We use the @Input() decorator to pass data from the parent to the child component via the template of the child component.

#### Parent component

For example, look at the following parent component:

import { Component } from '@angular/core';  
  
@Component({  
  selector: 'app-parent-comp',  
  template: `<app-child-comp [messageForChild]="parentMessage"></app-child-comp>`,  
  styleUrls: ['./parent-comp.component.scss']  
})  
export class ParentCompComponent {  
    
  parentMessage = "Tidy your room";  
  constructor() { }  
}

Here, we are using an **inline template**, which has the child component’s tag passing the parentMessage property to the child component via the messageForChild property.

#### Child component

Let’s look at the child component. In this example code you can see that we’ve set the @Input() decorator on the messageForChild property. Now, this property is recognized by Angular as a way to pass properties into the child component.



## Passing data from the child component to the parent component

### @ViewChild() decorator

There are different ways that a child component can pass data to the parent. One way is through using the @ViewChild() decorator. This decorator is used in the parent component to define a property that is a reference to the child component.

#### Parent component

Using our parent/child components from the previous example, we could add a reference in the ParentComponent like this:

import { Component, ViewChild, AfterViewInit } from "@angular/core";  
import { ChildCompComponent } from "../child-comp/child-comp.component";  
  
@Component({  
  selector: 'app-parent-comp',  
  template: `  
  Message: {{ message }}  
  <app-child-comp></app-child-comp>`,  
  styleUrls: ['./parent-comp.component.scss']  
})  
export class ParentCompComponent implements AfterViewInit {  
  @ViewChild(ChildCompComponent) childComp;  
  constructor() {}  
  message: string;  
  ngAfterViewInit() {  
    this.message = this.childComp.message;  
  }  
}  

Here, we are using the @ViewChild() decorator to set the childComp property as the reference to ChildCompComponent. Thus making it and its public properties available to the parent. Then, in the ngAfterViewInit() life cycle hook, we’re setting the parent’s message property to be the same as the child’s message property.

### Using @Output decorator and EventEmitter class

The second way data can be passed from the child component to the parent is by using the @Output() decorator and the EventEmitter class. **This approach involves defining events that can be subscribed to**. The child component defines what events it emits, and the parent listens for these events, which can contain data.

This approach is ideal for when the child component needs to let the parent component know of button clicks or when form fields are changed within the child component.

#### Parent component

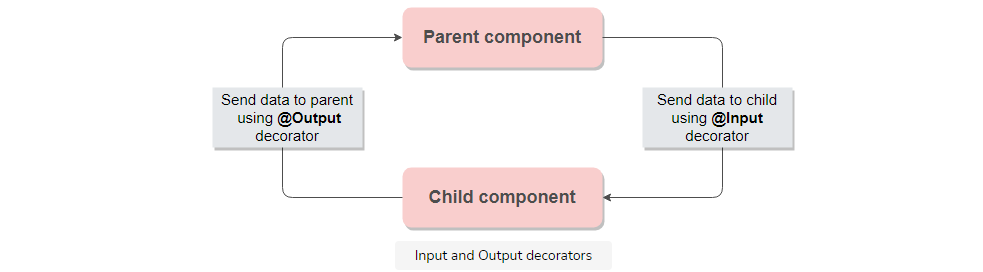
Here’s our parent component:

import { Component} from "@angular/core";  
  
@Component({  
  selector: 'app-parent-comp',  
  template: `  
  <app-child-comp  
      (messageEvent)="receiveMessageFromChild($event)">  
  </app-child-comp>`,  
  styleUrls: ['./parent-comp.component.scss']  
})  
export class ParentCompComponent {  
  
  constructor() {}  
  message: string;  
  receiveMessageFromChild($event) {  
    this.message = $event;  
    console.log (this.message)  
  }  
}  

In the parent, we’ve created an event handler called receiveMessageFromChild($event), which takes in a $event object. In the template, you can see that when messageEvent from the child is fired, the receiveMessageFromChild($event) handler is fired.

#### Child component

To create messageEvent in the child component, we’re using the @Output() decorator (the opposite of the @Input() decorator) to define an output, which is sent using EventEmitter. The <string> part of EventEmitter is saying that the message/data being passed in the event should be a type of string. This is another way TypeScript does type checking.



### Which approach is better?

So, out of these two approaches for sending data from child components to parent components, which is better? Well, it depends. The first approach clearly sees that **this child component is a child of the parent component; it is referenced within**. There is a **tighter cohesion between the two**.

The second approach, where the child component sends out events that the parent or any other component can listen for, **makes the relationship between the components more separated**, leading to **more flexibility if a codebase needs refactoring.** It also leads to more **separation of concerns**.

Using the approach of passing data from parents to child components using the @Input() decorator and data from the child to the parent component using events makes the flow of data in our application **one-way data flow**.

What really matters is that whatever approach you choose, you have to be consistent throughout the application.

# Passing Data Between Unrelated Components

Let's explore how we can pass data between unrelated components, i.e., components that don't have a parent/child relationship.

**We'll cover the following**

* [Using shared services](https://www.educative.io/courses/getting-started-with-angular/m7Q0Vk5Gnmp#Using-shared-services)
* [First component](https://www.educative.io/courses/getting-started-with-angular/m7Q0Vk5Gnmp#First-component)
* [Second component](https://www.educative.io/courses/getting-started-with-angular/m7Q0Vk5Gnmp#Second-component)

The final way of passing data between components is used for components that are unrelated – that is, components that don’t have this parent/child relationship.

## Using shared services

Passing data between unrelated components can be done through **shared services**. The following is a service that contains some data:

import { Injectable } from "@angular/core";  
  
@Injectable()  
export class DataService {  
  private message: string;  
  constructor() {}  
  changeMessage(newMessage: string) {  
    this.message = newMessage;  
  }  
  getMessage(): string {  
    return this.message;  
  }  
}

This service simply contains a private property called message, which can be changed by calling the changeMessage() function and retrieved using the getMessage() function.

The important part of this service is the @Injectable decorator. This tells Angular that this class can be injected into other classes, both components, and services.

We will be looking at services in more detail in [Chapter 8: Dependency Injection, Services, and HttpClient](https://www.educative.io/collection/page/10370001/4603693004488704/5343576383815680). But for now, all you need to know is that this class is injectable into other classes.

## First component

This is an example of a component using this service:

@Component({  
  selector: "app-parent-comp",  
  template: ` {{ message }} `,  
  styleUrls: ["./sibling.component.css"],  
})  
export class ParentComponent implements OnInit {  
  message: string;  
  constructor(private data: DataService) {}  
  ngOnInit() {  
    this.message = this.data.getMessage();  
    console.log(this.message);  
  }  
}

Here, we are injecting DataService using the constructor of the component class, then calling the DataService’s getMessage() function to set the component’s message property to the value in DataService.

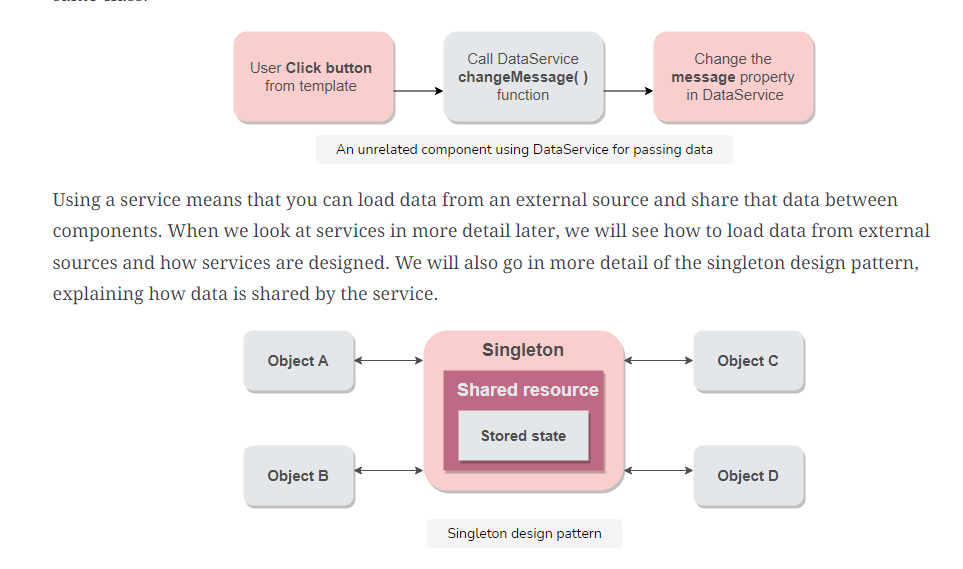
## Second component

Here is another example of the component using this service:

import { Component, OnInit } from "@angular/core";  
import { DataService } from "../data.service";  
  
@Component({  
  selector: "unrelated-comp",  
  template: `  
    {{ message }}  
    <button (click)="newMessage()">New Message</button>  
  `,  
  styleUrls: ["./sibling.component.css"],  
})  
export class UnrelatedComponent implements OnInit {  
  message: string;  
  constructor(private data: DataService) {}  
  
  ngOnInit() {  
    console.log(this.data.getMessage());  
  }  
  
  newMessage() {  
    this.data.changeMessage("Hello from a unrelated component");  
  }  
}

In this unrelated component, we are also injecting DataService, but when the user clicks the button from the template, the changeMessage() function of DataService is called, passing on a new message.

This will then update the message property in DataService, changing the value that is available to our original parent component. The reason that this data is the same for both components is that the service in **Angular is using a singleton design pattern**. This means that both components are accessing the same data from the same class.



# Create Template-driven Search Form for Application

Let's create a template-driven search form for our Client Contact Manager application.

**We'll cover the following**

* [Why use forms?](https://www.educative.io/courses/getting-started-with-angular/B6zwL3qZ0pY#Why-use-forms?)
* [Types of forms](https://www.educative.io/courses/getting-started-with-angular/B6zwL3qZ0pY#Types-of-forms)
* [Create a search form](https://www.educative.io/courses/getting-started-with-angular/B6zwL3qZ0pY#Create-a-search-form)
  + [Step 1: Create a new folder](https://www.educative.io/courses/getting-started-with-angular/B6zwL3qZ0pY#Step-1:-Create-a-new-folder)
  + [Step 2: Create a search-form component](https://www.educative.io/courses/getting-started-with-angular/B6zwL3qZ0pY#Step-2:-Create-a-search-form-component)
  + [Step 3: Update search-form.component.html](https://www.educative.io/courses/getting-started-with-angular/B6zwL3qZ0pY#Step-3:-Update-search-form.component.html)
  + [Step 4: Update search-form.component.ts](https://www.educative.io/courses/getting-started-with-angular/B6zwL3qZ0pY#Step-4:-Update-search-form.component.ts)
  + [Step 5: Update search-form.component.scss](https://www.educative.io/courses/getting-started-with-angular/B6zwL3qZ0pY#Step-5:-Update-search-form.component.scss)
  + [Step 6: Update app.module.ts file](https://www.educative.io/courses/getting-started-with-angular/B6zwL3qZ0pY#Step-6:-Update-app.module.ts-file)
  + [Step 7: Add new component to ClientPageComponent](https://www.educative.io/courses/getting-started-with-angular/B6zwL3qZ0pY#Step-7:-Add-new-component-to-ClientPageComponent)
* [Template-driven search form in action](https://www.educative.io/courses/getting-started-with-angular/B6zwL3qZ0pY#Template-driven-search-form-in-action)
* [Parent/Child relationship between components](https://www.educative.io/courses/getting-started-with-angular/B6zwL3qZ0pY#Parent/Child-relationship-between-components)

## Why use forms?

Forms are an extremely important part of any modern web application. Forms are used for **logging into an application, updating a user profile, submitting an application**, and so on. In our Client Contacts Manager application, we will be using forms to add new Clients and new Companies to it.

## Types of forms

Angular provides two types of forms:

* Reactive forms
* Template-driven forms

In many ways, both types work in the same way:

* They can both be used in the UI
* They allow the user to validate input fields

\*They both provide a model of the data being added via the form

But it’s how each form type handles and processes the data that’s submitted through the form that separates the two types.

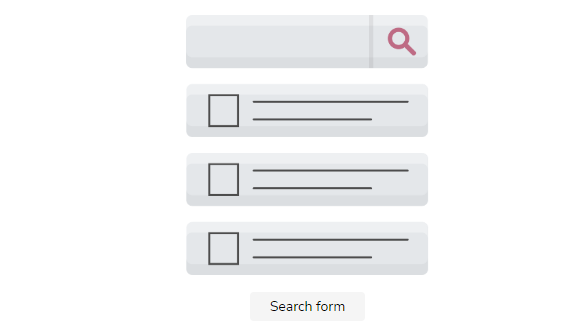
The template-driven form is probably the one you’ve already used, even if you haven’t used a JavaScript framework before. **An example of a template-driven form could be something like a simple login form or a small sign up form**.

Reactive forms use the Reactive programming approach, which is something we will explore more in the RxJs section. **Reactive forms are reusable, testable, and scalable** because of this Reactive programming approach.

In order to see the differences between the two types of forms, it’s best to go through a working example. In our Client Contacts Manager application, we are going to start adding two forms to the application. One is a simple template-driven **search form**, while the other is an **add Client form**.

## Create a search form

This simple search form will be used in the Clients section of our application. It will allow the user to **search for a client using the client’s last name**. However, before we start building out the template form, we need to make some changes to our application.

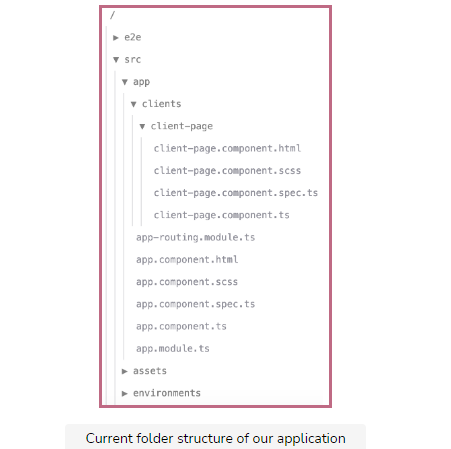
The first thing we need to do is create our new search form **presentational component**. Let’s go back to the Angular CLI, which will do this for us.

We’re not going to have all our components in the same folder. Instead, we’re going to start creating folders per functionality.

📝 **Note:** To get an idea of how you can structure your application, Angular has provided a [Style Guide](https://angular.io/guide/styleguide) that’s worth looking through.

### Step 1: Create a new folder

In our current folder structure, we have a Clients section, and within that we have a client-page component.

What we are going to do is create a new folder for our search form above the Clients section. To do this, we need to navigate to the app folder and create a new folder called search.

### Step 2: Create a search-form component

Now that we’ve created this folder, we need to navigate to it in the terminal.

cd search

Once you have navigated to this folder, run the following command to generate a component called search-form:

ng generate component search-form

### Step 3: Update search-form.component.html

Now, within the search folder, we have our new component. Let’s open up the HTML file of this component and make some changes by adding in a simple form. By default, the Angular CLI creates a template with just a paragraph tag and the title of the newly generated component. So our search-form template currently looks like this:

<p>  
       search-form works!  
</p>

Let’s update this template a bit. Here’s our updated search form template:

<form class="search-form" (ngSubmit)="onSubmit()">  
    <p><label for="searchBox">Search</label></p>  
    <input type="text" id="searchBox" placeholder="Search...." name="searchField" [(ngModel)]="searchField" />  
    <p><button type="submit">Submit</button></p>  
</form>

What we have done here is **added a new input field** called searchBox. We’ve also added a **submit button** and a **label** for this new form, but you’ll also see we introduced two new Angular-specific attributes. We’ve added (ngSubmit) and [(ngModel)], which are two new Angular attributes we’ve not yet encountered.

### Step 4: Update search-form.component.ts

Now, we will make some updates to our TypeScript component class:

import { Component, OnInit } from "@angular/core";  
  
@Component({  
  selector: "app-search-form",  
  templateUrl: "./search-form.component.html",  
  styleUrls: ["./search-form.component.scss"],  
})  
export class SearchFormComponent implements OnInit {  
  searchField: string;  
  
  constructor() {}  
  
  ngOnInit() {}  
  
  onSubmit() {  
    console.log(this.searchField);  
  }  
}

As you can see, there are a couple of changes here. The first change was made to the searchField property. This is a simple string, but through the use of [(ngModel)], we have bound the input field discussed in the previous section to this property. This is a one-way data-binding to the searchField property. Now, when we type anything into this text field it is set as the value of searchField.

The second change we’ve made is that we’ve added a new event, the onSubmit() event, which we have set as the event that’s fired when the form is submitted. Through the (ngSubmit) event binding in our form, we’ve made it so this onSubmit() event is fired when the form is submitted.

In this onSubmit() event, we are just writing the value of the local searchField property to the browser console. When we submit the form, any value that’s written in the input field is displayed in the browser console.

### Step 5: Update search-form.component.scss

Now we will do the styling of our component like this:

.search-form {  
  padding-top: 20px;  
  padding-left: 0;  
  label {  
    font-size: 16px;  
  }  
  input {  
    width: 600px;  
    height: 20px;  
    font-size: 14px;  
  }  
}

### Step 6: Update app.module.ts file

If we copy the above code into our application, we’ll find that there are some errors when we run the application. This is because we need to add FormsModuleto our application. Angular uses modules in the core framework to improve functionality. And FormsModule contains all the classes we need to add support to our application for both the template-driven forms and the Reactive forms.

📝 **Note:** This concept of adding modules to our application in order to add new functionality is something we had seen already when we added Angular Materials to our application in the [Getting Started With the Angular CLI chapter](https://www.educative.io/collection/page/10370001/4603693004488704/4539733223931904). It is also something we will explore further in the [NgModules chapter](https://www.educative.io/collection/page/10370001/4603693004488704/4518520179130368" \t "_blank) chapter.

Here is the updated app.module.ts file with the forms module imported:

import { BrowserModule } from "@angular/platform-browser";  
import { NgModule } from "@angular/core";  
import { FormsModule } from "@angular/forms";  
import { AppRoutingModule } from "./app-routing.module";  
import { AppComponent } from "./app.component";  
import { BrowserAnimationsModule } from "@angular/platform-browser/animations";  
import { ClientPageComponent } from "./clients/client-page/client-page.component";  
import { SearchFormComponent } from "./search/search-form/search-form.component";  
  
@NgModule({  
  declarations: [AppComponent, ClientPageComponent, SearchFormComponent],  
  imports: [  
    BrowserModule,  
    AppRoutingModule,  
    BrowserAnimationsModule,  
    FormsModule,  
  ],  
  providers: [],  
  bootstrap: [AppComponent],  
})  
export class AppModule {}

You can see that we’ve added the FormsModule to the imports array and where FormsModule was imported into the app.module.ts. By adding this to the imports array, Angular knows that this module and the functionality it contains is available to our application.

### Step 7: Add new component to ClientPageComponent

The last thing we need to do is add this new component to ClientPageComponent, like this:

<app-search-form></app-search-form>

## Template-driven search form in action

These are the basics of a template-driven form – we simply create a model (in this case, a simple string), which is bound to our form using the ngModel to set our local model to the form. Then, anything that’s submitted through this form is written to the model we create.

The idea of having a model behind the form is that it’s also used in a Reactive form, but how the form works with this model differs in the Reactive form, which we will explore in the next section.

## Parent/Child relationship between components

If we look at our Client Contacts Manager application, we can see an example of a **smart component** and a **presentational component**. Let’s go to the Clients section:

src > app > clients > client-page

In this folder, we have the following:

* client-page.component.html
* client-page.component.ts
* client-page.component.spec.ts
* client-page.component.sass

Now, if we open the template file of this component (client-page.component.html file), we’ll see that there isn’t much here, except a single presentational component:

<p>client-page works!</p>  
<app-search-form></app-search-form>

This presentational component adds the search functionality to our application. Here, we have started creating the parent/child relationship with our client-page as the parent component of the search-form component.

# Create Reactive Add Client Form for Application

Let's create a reactive form, where the user adds a new client to the Client Contacts Manager application.

**We'll cover the following**

* [Create client-form](https://www.educative.io/courses/getting-started-with-angular/YV7RV78PL6M#Create-client-form)
  + [Step 1: Create a client-form component](https://www.educative.io/courses/getting-started-with-angular/YV7RV78PL6M#Step-1:-Create-a-client-form-component)
  + [Step 2: Update app.module.ts](https://www.educative.io/courses/getting-started-with-angular/YV7RV78PL6M#Step-2:-Update-app.module.ts)
  + [Step 3: Update client-page.component.html](https://www.educative.io/courses/getting-started-with-angular/YV7RV78PL6M#Step-3:-Update-client-page.component.html)
  + [Step 4: Update client-form.component.html](https://www.educative.io/courses/getting-started-with-angular/YV7RV78PL6M#Step-4:-Update-client-form.component.html)
  + [Step 5: Update client-form.component.ts](https://www.educative.io/courses/getting-started-with-angular/YV7RV78PL6M#Step-5:-Update-client-form.component.ts)
  + [Step 6: Update client-form.component.scss](https://www.educative.io/courses/getting-started-with-angular/YV7RV78PL6M#Step-6:-Update-client-form.component.scss)
* [Reactive add client form in action](https://www.educative.io/courses/getting-started-with-angular/YV7RV78PL6M#Reactive-add-client-form-in-action)

So, how do Reactive forms differ from template-driven forms? As we’ve seen, the template-driven form is closely bound to its model. The search form in the [previous lesson](https://www.educative.io/collection/page/10370001/4603693004488704/4914670983970816) had a simple model of a string property, and as soon as we submit the form, that property is set. Reactive forms differ from template forms through their **use of observables to stream form data to the model**, while the template form is bound directly to the form.

In order to show the differences between the two approaches, it’s best to create a Reactive form so that we can look at the two side-by-side. Let’s start by creating a new form for our application.

## Create client-form

The form we’re going to create is our client-form, where the user will **enter the contact details of the new client** they are adding to the Client Contacts Manager application.

Add client form

### Step 1: Create a client-form component

The first thing we need to do is create a component that will contain this form. This component will be within our clients folder of the application and not in a separate folder like the Search form is.

Within the terminal, let’s navigate to the client folder and run the following command:

ng generate component client-form

This will create our new component, and it will be ready for the new form to be built.

### Step 2: Update app.module.ts

Before we start building out the HTML of the form, we need to do two more things. First, we need to add a new module to our app.module.ts file. This new module is called ReactiveFormsModule.

In the imports array of our app.module.ts file, we simply add **ReactiveFormsModule**. It will look like this:

imports:[  
    BrowserModule,  
    AppRoutingModule,  
    BrowserAnimationsModule,  
    FormsModule,   
    ReactiveFormsModule  
],

Then add the following import statement for this module to the top of the app.module.ts file:

import { FormsModule, ReactiveFormsModule } from '@angular/forms';

### Step 3: Update client-page.component.html

Now, we need to add this new component to ClientPageComponent, like this:

<h2>Add a new Client</h2>  
<app-search-form></app-search-form>  
<app-client-form></app-client-form>

We’ve also changed the title of the page.

### Step 4: Update client-form.component.html

Now, let’s open the template of our new client-form component and build out the HTML of the template:

<div class="client-form">  
  <div class="client-form">  
    <form [formGroup]="clientForm" (ngSubmit)="saveClient()">  
      <p>  
        <label> Firstname </label>  
        <input type="text" name="firstnameTxt" formControlName="firstname" required />  
      </p>  
      <p>  
        <label> Lastname </label>  
        <input type="text" name="lastnameTxt" formControlName="lastname" required />  
      </p>  
      <p>  
        <label> Email </label>  
        <input type="email" name="emailTxt" formControlName="email" required />  
      </p>  
      <p>  
        <label> Telephone </label>  
        <input type="tel" name="telephoneTxt" formControlName="telephoneNumber" />  
      </p>  
      <p>  
        <label> Company Name </label>  
        <input type="text" name="companyTxt" formControlName="companyName" required />  
      </p>  
      <p><button type="submit" [disabled]="!clientForm.valid">Save</button></p>  
    </form>  
  </div>  
</div>

At first glance, this Reactive form does look similar to the template form, but there are some slight differences. First, you can see that we’ve created something called a formGroup and added a formControlName attribute to each input element.

formGroup allows Angular to keep track of all the changes that happen to any of the formControl elements in our form. You can also see that we are using a property of the formGroup in the submit button. We’re checking to see if the form is valid, that it has passed all our checks before the Submit button is enabled. This stops the user from being able to submit the form before all the required fields have been completed.

### Step 5: Update client-form.component.ts

Now, let’s look at the Component class for this template, client-form.component.ts:

import { Component, OnInit } from '@angular/core';  
import {  
  Form,  
  FormBuilder,  
  FormGroup,  
  FormControl,  
  Validators  
} from '@angular/forms';  
  
@Component({  
  selector: 'app-client-form',  
  templateUrl: './client-form.component.html',  
  styleUrls: ['./client-form.component.scss']  
})  
export class ClientFormComponent implements OnInit {  
  // new FormGroup is defined here  
  clientForm: FormGroup;  
  
  // creating new FormControls, with validation  
  firstname = new FormControl('', Validators.required);  
  lastname = new FormControl('', Validators.required);  
  email = new FormControl('', Validators.required);  
  telephoneNumber = new FormControl('');  
  companyName = new FormControl('', Validators.required);  
  
  // in the constructor we create a FormGroup and set the properties of  
  // the formGroup to the FormControls then set it to be the clientForm we use  
  // in the template.  
  constructor(fb: FormBuilder) {  
    this.clientForm = fb.group({  
      firstname: this.firstname,  
      lastname: this.lastname,  
      email: this.email,  
      telephoneNumber: this.telephoneNumber,  
      companyName: this.companyName  
    });  
  }  
  
  ngOnInit() {}  
  
  // event called when form is submitted, displaying the output of the form  
  saveClient() {  
    console.log(this.clientForm);  
  }  
}

So, what’s happening here? Well, we created a new FormGroup object called clientForm, which is exposed as public property to the template. Then, we created a series of FormControl objects, one for each form field, to set the default value (currently an empty string) and the validation options, which required.

Then, in the constructor, we created a new FormBuilder instance and set the FormControl objects as properties of the new Group that we created using the FormBuilder.

Finally, we have an event, saveClient(), which is called when the form is submitted. This event simply outputs the values of our form fields, which are stored in the clientForm object so that we can see them in the browser console.

But what does all this mean? Well, let’s start with FormControls. We’ve made a series of them: one for each of our form’s input elements. By doing this, we are telling Angular about these form elements, and Angular can now observe these form fields. Angular can check what values have been added to them if they pass the validation rules that have been set.

Next, we use Angular’s FormBuilder class to create a new group object. This group is a representation of the values entered in the form fields, similar to how we used ngModel in the template-driven form. This group allows us to get the values out of the form. It then creates the model of the form. Then, we set this model representation of the form to the clientForm public property, which we then set as the formGroup of the template.

When the form is submitted, all the values of the form fields are set to this clientForm object. So when we write the clientForm object to the browser console, we see an object with all the form values as properties of this FormGroup object.

### Step 6: Update client-form.component.scss

Now, you should style the component as:

.client-form {  
  padding-top: 20px;  
  p {  
    font-size: 1.1em;  
  }  
  button {  
    font: 1em sans-serif;  
  }  
}

## Reactive add client form in action[**#**](https://www.educative.io/courses/getting-started-with-angular/YV7RV78PL6M#Reactive-add-client-form-in-action)



# When to Use Reactive and Template Forms

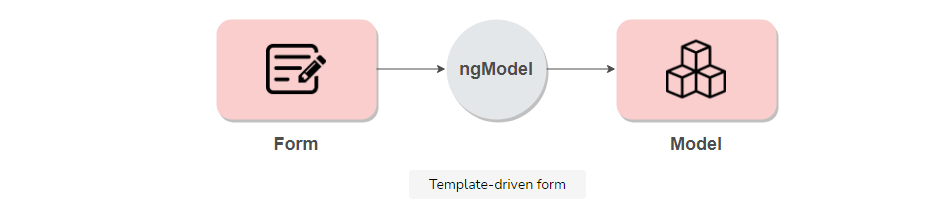
Let's explore the differences between the reactive and template forms. We’ll also look at the reasons why we could use either.

**We'll cover the following**

* [Difference between reactive and template-driven form](https://www.educative.io/courses/getting-started-with-angular/m79OAKKm0Pr#Difference-between-reactive-and-template-driven-form)
* [When to use template forms](https://www.educative.io/courses/getting-started-with-angular/m79OAKKm0Pr#When-to-use-template-forms)
* [When to use reactive forms](https://www.educative.io/courses/getting-started-with-angular/m79OAKKm0Pr#When-to-use-reactive-forms)

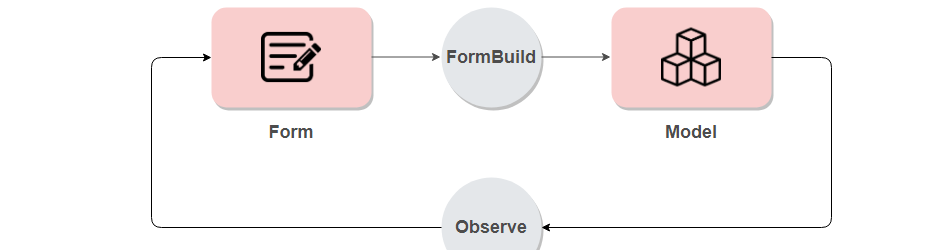
## Difference between reactive and template-driven form

The difference between the template-driven form and the Reactive form is that the **template form binds closely to a model through ngModel**. As the form is submitted, the model is already set.

Reactive forms take a different approach – they create a model of the form, which is attached to the form using the *FormGroup*. This model then observes the form, and as data is entered into the form, the model updates. This makes the state of the model available at any time so that if the user has just entered the first part of the form, we have a model representation of this. With each change to the form, we have a new instance of our form model.

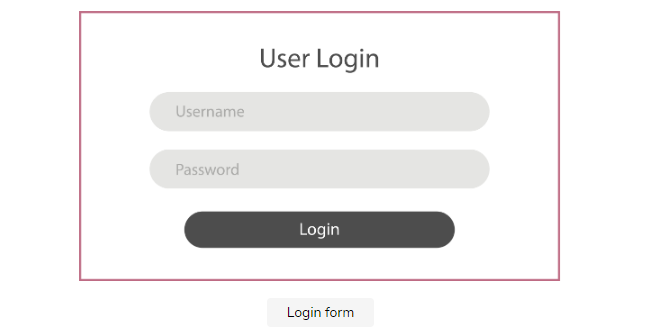
This means we can start taking advantage of the Reactive programming model which Angular now embraces, instead of waiting for the user to complete and submit the form as we do in the template-driven approach. Through this Reactive approach, we can start using the data in the form as the user types. You can see an example in the [previous lesson](https://www.educative.io/collection/page/10370001/4603693004488704/6258237853663232). In the form’s Valid attribute, we have added the check for the Submit button. The form is being checked as the user types into the form, and once all the required fields have been completed, the Valid attribute of the form is set to true.

We can **start checking the value of a form field as the user types**. This is helpful for forms where we need to start filtering data based on the value being entered, for example, a form field that filters a list of companies. It sounds like we need to start building out our application so that we can add a reactive form to our **Company** section.



## When to use template forms

We would use a Template form when we need to **build a simple form**, or our project has the need for a simple form as part of simple scenarios. Template forms are ideal for simple forms, such as login forms.

If your project is not heavily reliant on Unit Testing, then template forms are an option. **Template forms are hard to test**, but if that’s not an issue for your project, then template forms are an option.

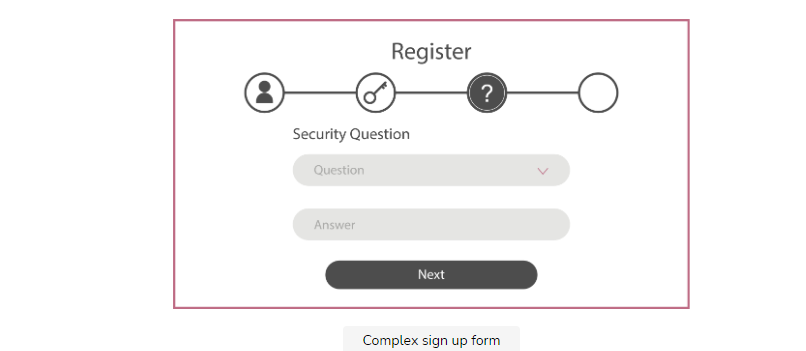
With a template form, all the **data handling is managed by Angular**, and two-way binding is handled using the [(ngModel)] syntax. As we’ve seen in the examples so far, the TypeScript code needed for a template form is far less complex than it is for a Reactive form.

Generally, template forms are ideal for small, simple forms. They are similar to how forms were handled in AngularJS but not ideal for more complex forms, which a lot of modern web applications use.

## When to use reactive forms

Reactive forms offer a lot more features, and they make testing far easier. If your **project requires good testing coverage** (when all the code that’s been written has an associated unit test), then Reactive forms are a better option.

They are **ideal for more complex scenarios in applications**. With Reactive forms, we can build complex signup wizards where each page of the form is a new step of the signup process. With Reactive forms, new form elements can be added dynamically. So, if a user selects an option, we can change the next step in the process based on the value of a previous option.

There is **no data binding**, so we can make an immutable data model of the form that we can work with. This produces more TypeScript code but less HTML code. This is something to consider if you are happier working in TypeScript than HTML.

Reactive forms are hard to use at first, but with practice, you’ll find they soon become your default choice for creating forms in Angular. They offer so much more functionality, flexibility, and ways of handling data that it is worth investing the time to understand them.

# NgModules

Let's learn about Ng modules in this chapter.

**We'll cover the following**

* [Chapter learning outcomes](https://www.educative.io/courses/getting-started-with-angular/N0XmXlLqqqL#Chapter-learning-outcomes)
* [Client contacts manager application](https://www.educative.io/courses/getting-started-with-angular/N0XmXlLqqqL#Client-contacts-manager-application)

## Chapter learning outcomes

So far, we’ve discussed the architecture of an Angular project. We’ve created the basic version of our demo application, the Client Contacts Manager, and added some components to this application. Everything we’ve done so far has all been developed within the single module that Angular generates for us.

In this chapter, we will start looking further into **modules in Angular**. We had already seen new modules being used when we added Angular Material and Forms to our example application, but now we are going to look into how modules are structured, what each part of the module file does, and how to **create them using the Angular CLI**.

We will then look at **why modules are used** and how they help divide up the application’s functionality, especially for large enterprise applications. We will then continue expanding on our example application by adding new modules for the various sections of our application. This chapter will help you to develop your own Angular application modules. And it will show you **how you can use modules to structure your application**.

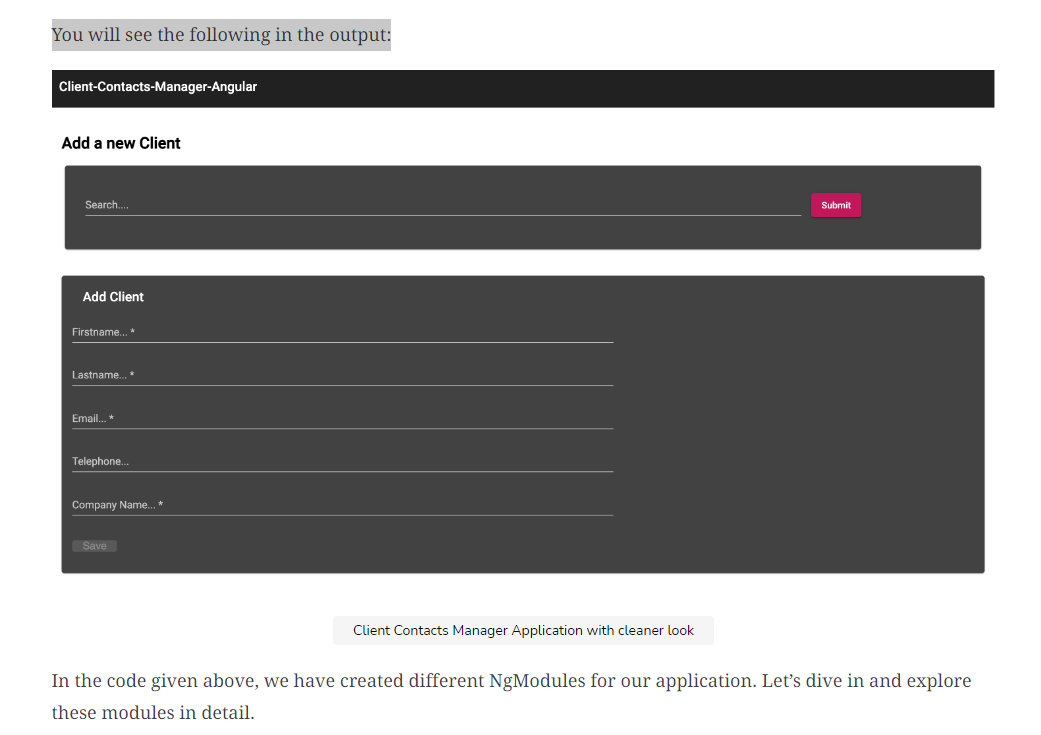
By the end of this chapter, you will learn the following:

* How modules are structured
* How to add your components to a module
* How to create modules using the CLI
* How to separate the functionality of an application using modules
* How to plan the structure of applications using modules

## Client contacts manager application

📝 **Note:** Here is the preview of what we will create in this chapter. Press the **RUN** button to compile and serve the application. Once the app compiles, you can see its output in the **Output** tab or by clicking on the URL given after Your app can be found at.





# What are Modules in Angular?

Let's explore what modules are in Angular and how we can start an Angular application.

**We'll cover the following**

* [Modules?](https://www.educative.io/courses/getting-started-with-angular/mEQOEv7MkxE#Modules?)
* [Structure of a module](https://www.educative.io/courses/getting-started-with-angular/mEQOEv7MkxE#Structure-of-a-module)
* [How Angular starts an application?](https://www.educative.io/courses/getting-started-with-angular/mEQOEv7MkxE#How-Angular-starts-an-application?)
* [Root module](https://www.educative.io/courses/getting-started-with-angular/mEQOEv7MkxE#Root-module)
* [Angular/core module](https://www.educative.io/courses/getting-started-with-angular/mEQOEv7MkxE#Angular/core-module)

## Modules?

**Modules** are the glue that holds an application together. They are single TypeScript files that reference all the other files used within the application. They allow us, as Angular developers, to group the functionality of our application together.

From the AngularJS days, modules have been a core part of an Angular application. In Angular 2+, a new way of creating modules was introduced. The new @NgModules class was introduced as the new way to create modules of functionality in an application.

This new approach allows us as Angular developers to group together the elements of the module (the components, services, pipes, and directives) under a separate module. Then, we can group individual modules together to make a complete application. It’s a lot like how pieces of a puzzle are put together to make a complete picture.

Angular groups individual modules together to make a complete application

## Structure of a module

The NgModule class allows us to create a public API of the components available in the module. In the NgModule class, we export the elements (the Components or Services) of the module so these elements can be accessed by other modules in our application.

Let us take a look at an example of a @NgModule class so we can explore how a module is structured:

import { NgModule } from '@angular/core';  
import { BrowserModule } from '@angular/platform-browser';  
import { AppComponent } from './app.component';  
  
@NgModule({  
     imports: [BrowserModule],  
     declarations: [AppComponent],  
     bootstrap: [AppComponent]  
})  
export class AppModule { }

In this app.module.ts file, we’ve created a module that imports the Angular BrowserModule, declares the AppComponent, and sets the same AppComponent as the first component to run when the application starts.

📝 **Note:** The BrowserModule is part of the Angular framework. It provides all the functionality needed by the framework to start an application running in the browser. For more information on this module, check out the official [Angular docs](https://angular.io/api/platform-browser/BrowserModule).

## How Angular starts an application?

The main.ts file, which is a TypeScript file the Angular CLI creates as part of an application that tells Angular what module to call when booting up/starting the application:

###### /

e2e

src

app

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.scss

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**main.ts**

1

2

3

4

5

6

7

8

9

10

11

12

13

import { enableProdMode } from '@angular/core';

import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';

import { environment } from './environments/environment';

if (environment.production) {

  enableProdMode();

}

platformBrowserDynamic().bootstrapModule(AppModule)

  .catch(err => console.error(err));





Run

Save

Reset

**Your app can be found at:**[https://ed-6434592358924288.educative.run](https://ed-6434592358924288.educative.run/)

main.ts file in Angular

In the preceding example, we can see that the platformBrowserDynamic class of the Angular platform has been imported, and then almost immediately, the ***bootstrapModule() function*** of this class is called with the *AppModule* class of our application passed in as the module that Angular will use to start our application.

This is how an Angular application starts. It needs one NgModule class to start the application. Then from there, all the other modules of our application are imported into AppModule for the framework to find and load when needed as the application runs in the browser.

## Root module

As you learn more about Angular, you’ll see other modules, such as the **root module**. This module handles all the **bootstrapping of the application**. Its role is to manage the startup of the application. If you look at the main.ts file of an Angular application, you’ll see the following code:

platformBrowserDynamic().bootstrapModule(AppModule).catch(err => console.error(err));

This is where the framework is taking the AppModule and setting it as the root module that will be run first, as part of this startup process.

## Angular/core module

If you look at our project, you’ll see that there are other modules within Angular to be imported or added to our own modules. These modules show how functionality is grouped together so that it can be imported or added to our own modules.

You’ll see that in the import statements, these are the lines of code at the top of our file, how different modules are added. For example, at the top of a Module file, you’ll see the following:

import { NgModule } from '@angular/core';

This is an import statement where we are adding the NgModule class from the angular/core module. This angular/core module is part of the Angular framework that contains a set of TypeScript classes, which **provide the core functionality of the Angular framework**. Creating a module is part of this core functionality, so this is part of the Angular/core module.

Angular makes use of modules, not only to group pieces of functionality together but also as a way to access and make use of the core functionality of the framework.

So the importance of understanding how modules work and can be used within an Angular application is extremely important.

# The Sections of the NgModule File

Let's explore the purpose of each array in the 'NgModule' file.

**We'll cover the following**

* [Arrays in @NgModule](https://www.educative.io/courses/getting-started-with-angular/3wV8ovK8yyx#Arrays-in-@NgModule)
* [The declarations array](https://www.educative.io/courses/getting-started-with-angular/3wV8ovK8yyx#The-declarations-array)
* [The imports array](https://www.educative.io/courses/getting-started-with-angular/3wV8ovK8yyx#The-imports-array)
  + [Example](https://www.educative.io/courses/getting-started-with-angular/3wV8ovK8yyx#Example)
* [The exports array](https://www.educative.io/courses/getting-started-with-angular/3wV8ovK8yyx#The-exports-array)
* [The providers array](https://www.educative.io/courses/getting-started-with-angular/3wV8ovK8yyx#The-providers-array)
* [The bootstrap array](https://www.educative.io/courses/getting-started-with-angular/3wV8ovK8yyx#The-bootstrap-array)
* [The entryComponents array](https://www.educative.io/courses/getting-started-with-angular/3wV8ovK8yyx#The-entryComponents-array)

## Arrays in @NgModule

When the Angular CLI creates a new application, it generates a single NgModule file for us. This file is always called app.module.ts and looks like this:

###### /

e2e

src

app

app-routing.module.ts

app.component.html

app.component.scss

app.component.spec.ts

app.component.ts

app.module.ts

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.scss

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**app.module.ts**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

import { BrowserModule } from '@angular/platform-browser';

import { NgModule } from '@angular/core';

import { AppRoutingModule } from './app-routing.module';

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

import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

@NgModule({

  declarations: [

    AppComponent

  ],

  imports: [

    BrowserModule,

    AppRoutingModule,

    BrowserAnimationsModule

  ],

  providers: [],

  bootstrap: [AppComponent]

})

export class AppModule { }





Run

Save

Reset

**Your app can be found at:**[https://ed-6434592358924288.educative.run](https://ed-6434592358924288.educative.run/)

app.module.ts file in Angular

Now let’s take a look at each of the sections of the @NgModule in more detail.

## The declarations array

The declarations section is one of the **static metadata sections**. In this, an array is defined where all the **components, directives, and pipes of this module are declared**.

The components, directives, and pipes that are listed in this array can only **belong to one module**. If you add a component to more than one NgModule, the compiler will throw an error.

###### Question

How can we share a component in several different modules of our application?

Show Answer

By declaring the components, directives, and pipes in the declarations array, we’re telling Angular where it can **find the tag selectors** used throughout the application. Without this, Angular won’t know what templates to use when it encounters a selector in our templates, so it is a crucial part of the compilation process.

## The imports array

The imports array contains the **composition category of metadata**. This array contains the names of the other modules that can be included in this module, and it is the components, directives, or pipes that are exported by these modules that make them available in other modules.

Basically, the way components are shared between modules is by setting at the module level which of its components, directives, and pipes are set as exported. Then, when the module is imported into another module, these exported elements can be used in the module that has been imported into the other module.

### Example

Examples of this use of the imports array include the FormsModules, which we have already seen in the [Components, Templates, and Forms chapter](https://www.educative.io/collection/page/10370001/4603693004488704/6258237853663232). In that chapter, when we started adding both the template-driven form and the reactive forms to our Client Contacts Manager application, we imported the FormsModule into our AppModule. By doing this, we had access to all the components and directives that the FormsModule has set as accessible in its **exports** array.

AppModule has access to all the components directives of FormModule

## The exports array

The exports array is the other **composition metadata array**. Both this and the imports array are closely related.

The exports array contains the list of components, directives, and pipes that can be used by the importing module. Until this point, all the components are private within the module. When they are listed in the exports array, they are declared public.

If we think of this as a TypeScript class, all we are doing is setting what properties are public by setting them in the exports array. So when we make an instance of this TypeScript class, the public properties are available in the class that is using this new instance of our TypeScript class.

This is what we’re doing with the exports array. We’re telling the Angular compiler that the components, directives, and pipes declared in this array are public and accessible in other modules. But only if the other module includes the NgModule in which this export array is defined.

This means that not all of your components need to be added to the exports array, only the ones you want to be made available to other modules of the application.

## The providers array

The providers array is part of the **runtime metadata** of our NgModule decorator. In this array, we list all the providers that can be injected via Dependency Injection into our components.

💡 **Did you know?**

**Dependency Injection** is a way of passing objects/services to other objects instead of using a factory class to generate these new objects. With dependency injection, objects are passed into other objects when needed. We will be exploring dependency injection in the [Dependency Injection, Service, and HttpClient chapter](https://www.educative.io/collection/page/10370001/4603693004488704/5343576383815680).

By adding these providers to this providers array, we’re registering them with the module, and it makes them available to all of the components, directives, and pipes within this module. Without doing this, we could find that when we try to use one of these services later in a component, the compiler will throw an error, telling us that the service we want is not registered with the parent module.

The Angular compiler is extremely smart. It will let us know if we are trying to use a provider that isn’t registered, but it is better to remember to add our services to the providers array beforehand.

Dependency Injection

## The bootstrap array

This array is another **static type of metadata**. It contains a list of the components that will be automatically bootstrapped/started when the compiler runs this module.

Usually, this bootstrap array has one component. As we can see from our app.module.ts file, in the bootstrap array of this module, we have just the AppComponent listed. This means that the AppComponent will be called when the module starts running.

###### Question

We did say that there is usually only one component listed in the bootstrap array, but is it possible to add more than one? Why would you do this?

Show Answer

## The entryComponents array

This array is another **static metadata array**, and it **contains a list of the components that can be dynamically loaded into the view**.

When an Angular application starts, it uses the component listed in the bootstrap array to start the application and load the bootstrap listed component into the view. Then, as the user navigates throughout the application, they move through a list of routes that we define. And as part of this route definition, we set what component is loaded when that route is run.

📝 **Note:** We will be going through routes and defining routes in the [Routing and Navigation chapter](https://www.educative.io/collection/page/10370001/4603693004488704/5920025016795136), where we will see how to define these components that run as part of a route.

So, we now know of two ways that a component is automatically loaded into the view, either when it’s set in the bootstrap array or when it’s defined in a route. For Angular to know of these components at the time the module is being compiled, Angular automatically adds these components to the entryComponents array.

# How to Create Modules Using the CLI

Let's explore how we can create a module using the Angular CLI.

**We'll cover the following**

* [Create the module in the current folder](https://www.educative.io/courses/getting-started-with-angular/YQ4gGYnl6vA#Create-the-module-in-the-current-folder)
* [Create a module within a specific folder](https://www.educative.io/courses/getting-started-with-angular/YQ4gGYnl6vA#Create-a-module-within-a-specific-folder)
* [Import submodules within AppModule](https://www.educative.io/courses/getting-started-with-angular/YQ4gGYnl6vA#Import-submodules-within-AppModule)
* [Modules as trees of components](https://www.educative.io/courses/getting-started-with-angular/YQ4gGYnl6vA#Modules-as-trees-of-components)

## Create the module in the current folder

So we’ve covered the metadata of the @NgModule decorator, and now it’s time to start creating our own modules. Now that we know the structure of a NgModule class, we could start writing our modules by hand, but thankfully, the Angular CLI team has created a way to generate modules with a few simple commands.

To create a new module using the CLI, we use the generate command. We’ve already used this command to generate new components in our example application. The full command for creating a module is as follows:

ng generate module <modulename>

So if we wanted to create a new module named **AdminModule**, we would simply use this command:

ng generate module AdminModule

Or use the shortcut format as follows:

ng g module AdminModule

## Create a module within a specific folder

While these commands will generate a new NgModule file for us, sometimes we may want to tell the CLI where to create this new module. For example, if we are creating this AdminModule, we may want it to be generated in a folder called admin. There are two ways we can do this. First, we can navigate to the admin folder within a terminal, then run the ng generate module command, or we can put the complete path before the name of the new module to be generated like so:

ng generate module admin/AdminModule

This will then generate the new module TypeScript file for us in the following:

src/app/admin/admin-module/admin-module.module.ts

As you can see, what the CLI has done is created a folder called the admin-module within the admin folder. The CLI uses the name we’ve provided for our module to create a unique folder name.

## Import submodules within AppModule

Now that we’ve created a new module, we can start adding components to this newly created module, but before we do, let’s recap on our folder structure.

If we create this new admin module in the angular-architecture project that we created in the [Angular Architecture chapter](https://www.educative.io/collection/page/10370001/4603693004488704/5437878287990784), our new folder structure will look like this:

###### /

e2e

src

app

admin

admin-module

admin-module.module.ts

app.component.css

app.component.html

app.component.spec.ts

app.component.ts

app.module.ts

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.css

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**admin-module.module.ts**

1

2

3

4

5

6

7

8

9

10

11

12

13

import { NgModule } from '@angular/core';

import { CommonModule } from '@angular/common';

@NgModule({

  declarations: [],

  imports: [

    CommonModule

  ]

})

export class AdminModuleModule { }





Run

Save

Reset

**Your app can be found at:**[https://ed-6434592358924288.educative.run](https://ed-6434592358924288.educative.run/)

Structure of Angular application

See now we have this new admin/admin-module. And inside that folder we have admin-module.module.ts, which is a very basic NgModule file, but it does the job.

Then in order for this new sub-module to be added to our main AppModule, we add it to the imports array like this in the app-module.ts file:

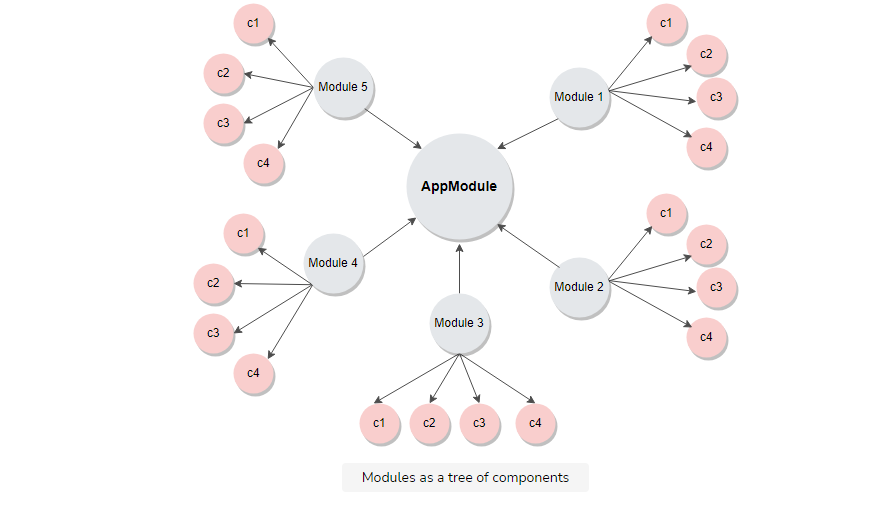
import { AdminModuleModule } from './admin/admin-module/admin-module.module';  
  
@NgModule({  
  declarations: [  
    AppComponent  
  ],  
  imports: [  
    BrowserModule,  
    AdminModuleModule  
  ],  
  providers: [],  
  bootstrap: [AppComponent]  
})

Then as we create new components within our new admin section, these components are added to the new AdminModule, which through being imported into the main AppModule, makes these “admin” components part of our final application.

## Modules as trees of components

A good way to think of this is seeing modules as trees of components, and each module is linked to the next, building up each individual tree module. **Angular can then watch this set of trees**, and it knows what each component is, where it is, and how to use the component when needed.

Modules as a tree of components



Being able to create modules so easily can allow us to start creating modules for everything. This is the beauty of the Angular CLI. It allows us to take repetitive tasks and make the simple one-line commands we can run to generate new code for our application. However, doing this means we need to start thinking about the structure of our applications and how we are going to start using modules.

# Creating Modules for Our Client Contact Manager Application

Let’s use the Angular CLI to create some more modules for our Client Contacts Manager application.

**We'll cover the following**

* [Sections of the application](https://www.educative.io/courses/getting-started-with-angular/B6DW0K3Z89n#Sections-of-the-application)
  + [Client section](https://www.educative.io/courses/getting-started-with-angular/B6DW0K3Z89n#Client-section)
  + [Company section](https://www.educative.io/courses/getting-started-with-angular/B6DW0K3Z89n#Company-section)
* [Modules of our application](https://www.educative.io/courses/getting-started-with-angular/B6DW0K3Z89n#Modules-of-our-application)
  + [AppModule](https://www.educative.io/courses/getting-started-with-angular/B6DW0K3Z89n#AppModule)
  + [ClientModule](https://www.educative.io/courses/getting-started-with-angular/B6DW0K3Z89n#ClientModule)
  + [CompanyModule](https://www.educative.io/courses/getting-started-with-angular/B6DW0K3Z89n#CompanyModule)
  + [SharedModule](https://www.educative.io/courses/getting-started-with-angular/B6DW0K3Z89n#SharedModule)
  + [CustomMaterialModule](https://www.educative.io/courses/getting-started-with-angular/B6DW0K3Z89n#CustomMaterialModule)
* [Create ClientModule](https://www.educative.io/courses/getting-started-with-angular/B6DW0K3Z89n#Create-ClientModule)
* [Create CompanyModule](https://www.educative.io/courses/getting-started-with-angular/B6DW0K3Z89n#Create-CompanyModule)
* [Update AppModule](https://www.educative.io/courses/getting-started-with-angular/B6DW0K3Z89n#Update-AppModule)
* [Client and Company modules in action](https://www.educative.io/courses/getting-started-with-angular/B6DW0K3Z89n#Client-and-Company-modules-in-action)

## Sections of the application

Now that we know how to use the Angular CLI to create modules, we can move forward with our Client Contact Manager application. Let’s review the application we’re building and see how we can use our knowledge of modules to start building out the functionality of the application.

Character creating modules for Client Contacts Manager application

The application we’re building consists of two main sections: Client and Company.

### Client section

In the Client section, the user will be able to do the following:

* View a list of their clients
* Add a new client
* Search for a client
* Edit a client’s details
* Delete a client

### Company section

While in the Company section, they will be able to do the following:

* View the list of companies
* Add a new company
* Search for a company
* Edit the details of a company
* Delete a company

Both use cases are very similar, but we can clearly see a need for two separate sections in the application. For example, The user may decide that they just want to add a new company. Or, the user might encounter a situation where they had to look up the contact details of their favorite client.

Now that we know the two main use cases for the application, we might think that we only really need to use two modules to structure our application: one @NgModule for the Client section and one @NgModule for the Company section. But there are actually more modules we will use in this application.

## Modules of our application

As we know, modules in Angular can be seen as a series of trees, and each tree has a set of components that branch out of the module. So for this application, we’d have a NgModule for the Client section, and we would also have a NgModule for the Company section, but we also need to have the main AppModule, which connects both the Client and Company modules together. Here is a list of the modules we will be creating:

Modules of Client Contacts Manager application

There’s five in total. Let’s go through each one and see what it will be doing within the application.

### AppModule

This is the **main module of our application**. It is the default module that the Angular CLI creates when it generates a new Angular application. It will have all the UI components needed to create the shell of the application.

### ClientModule

It is the **main module of the Client section**. This is a child module of the AppModule that contains all the components for the Client section UI.

### CompanyModule

It is the **main module of the Company section**. Again a child of the AppModule, it contains all the components for the Company UI.

### SharedModule

This module is used to **share components across both the Client and Contacts**.

### CustomMaterialModule

This module is used to **import modules we may need from the Angular Material** UI library.

Now, let’s start creating the ClientModule and CompanyModule of our application.

## Create ClientModule

We already have a clients folder we created when generating the client-page and client-form components during the [Components, Templates, and Forms chapter](https://www.educative.io/collection/page/10370001/4603693004488704/6258237853663232). We go to that folder and create a new TypeScript file called client.module.ts and add this code:

import { NgModule } from "@angular/core";  
import { CommonModule } from "@angular/common";  
@NgModule({  
  imports: [CommonModule],  
})  
export class ClientModule {}

## Create CompanyModule

Now we need to create the CompanyModule. First, under the app folder we need to create a new company folder under the app folder. Then inside the company folder, create a new TypeScript file called company.module.ts, and inside that file add the following code:

import { NgModule } from "@angular/core";  
import { CommonModule } from "@angular/common";  
@NgModule({  
  imports: [CommonModule],  
})  
export class CompanyModule {}

## Update AppModule

Great, now we have our two modules. We need to now add these modules to our app.module.ts in order to link these new modules with our app module. To do that, we add these modules to the imports array of app.module.ts as follows:

imports: [  
  BrowserModule,  
  AppRoutingModule,  
  BrowserAnimationsModule,  
  ClientModule,  
  CompanyModule,  
  FormsModule,  
  ReactiveFormsModule,  
];

## Client and Company modules in action

Now we’ve done a lot of new code additions to our application, let’s run the app to make sure everything is still running ok.

📝 **Note:** Press the **RUN** button to compile and serve the application. Once the app compiles, you can see its output in the **Output** tab or by clicking on the URL given after Your app can be found at.

###### /

e2e

src

app

company

clients

search

app-routing.module.ts

app.component.html

app.component.scss

app.component.spec.ts

app.component.ts

app.module.ts

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.scss

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**app.module.ts**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

import { BrowserModule } from '@angular/platform-browser';

import { NgModule } from '@angular/core';

import { FormsModule, ReactiveFormsModule } from '@angular/forms';

import { AppRoutingModule } from './app-routing.module';

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

import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { ClientModule } from './clients/client.module';

import { CompanyModule } from './company/company.module';

import { ClientPageComponent } from './clients/client-page/client-page.component';

import { SearchFormComponent } from './search/search-form/search-form.component';

import { ClientFormComponent } from './clients/client-form/client-form.component';

@NgModule({

  declarations: [

    AppComponent,

    ClientPageComponent,

    SearchFormComponent,

    ClientFormComponent

  ],

  imports: [

    BrowserModule,

    AppRoutingModule,

    BrowserAnimationsModule,

    ClientModule,

    CompanyModule,

    FormsModule,

    ReactiveFormsModule

  ],

  providers: [],

  bootstrap: [AppComponent]

})





Run

Save

Reset

**Your app can be found at:**[https://ed-6434592358924288.educative.run](https://ed-6434592358924288.educative.run/)

Client Contacts Manager application after adding client and company module



# Adding Our Client Components to the ClientModule

Let's structure our application into different functional areas by tidying up our components and new modules.

**We'll cover the following**

* [Update client.module.ts file](https://www.educative.io/courses/getting-started-with-angular/qAQ8PK9Bpw0#Update-client.module.ts-file)
  + [Add components](https://www.educative.io/courses/getting-started-with-angular/qAQ8PK9Bpw0#Add-components)
  + [Import modules](https://www.educative.io/courses/getting-started-with-angular/qAQ8PK9Bpw0#Import-modules)
* [Create a shared module](https://www.educative.io/courses/getting-started-with-angular/qAQ8PK9Bpw0#Create-a-shared-module)
  + [Step 1: Create shared.module.ts file](https://www.educative.io/courses/getting-started-with-angular/qAQ8PK9Bpw0#Step-1:-Create-shared.module.ts-file)
  + [Step 2: Update app.module.ts](https://www.educative.io/courses/getting-started-with-angular/qAQ8PK9Bpw0#Step-2:-Update-app.module.ts)
  + [Step 3: Update client.module.ts](https://www.educative.io/courses/getting-started-with-angular/qAQ8PK9Bpw0#Step-3:-Update-client.module.ts)
* [Shared module in action](https://www.educative.io/courses/getting-started-with-angular/qAQ8PK9Bpw0#Shared-module-in-action)

## Update client.module.ts file

We’ve created the ClientModule. We can now add all our Client components to this module to start structuring our application into the different areas of functionality. When we do this refactoring, we will encounter a couple of issues, but first, let’s make the changes. Then we can go through the problems we encountered and why they happened.

This is our new client.module.ts file as follows:

import { NgModule } from "@angular/core";  
import { CommonModule } from "@angular/common";  
import { ClientPageComponent } from "./client-page/client-page.component";  
import { ClientFormComponent } from "./client-form/client-form.component";  
import { ReactiveFormsModule } from "@angular/forms";  
  
@NgModule({  
  declarations: [ClientPageComponent, ClientFormComponent],  
  imports: [CommonModule, ReactiveFormsModule],  
})  
export class ClientModule {}

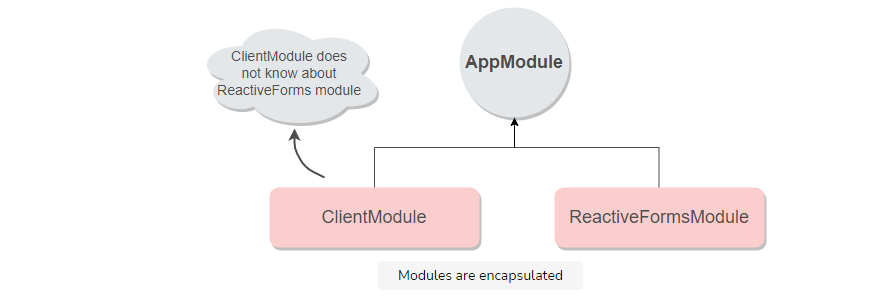
### Add components

As you can see, we’ve made a couple of changes. First, we’ve added the ClientPageComponent and the ClientFormComponent to the declarations array of the module. As we know, the declarations array is how we tell the ClientModule what components belong to it. So, when the ClientModule is imported into another module, say the AppModule, the AppModule knows what component, directive, or pipes belong to the ClientModule.

📝 **Note:** We will remove ClientPageComponent and the ClientFormComponent from the declarations array of the app.module.ts file.

### Import modules

Next, we have to import the ReactiveFormsModule. This is because we are using a reactive form in our Client form module. Remember **modules are encapsulated**, so when we add the ReactiveFormsModule to AppModule and then add the ClientModule to AppModule, it doesn’t mean that the ClientModule knows that ReactiveFormsModule is included in AppModule. ClientModule needs to have a direct connection to ReactiveFormsModules, and this is what we are doing by adding it into the ClientModule imports array.



If we look at the terminal, we will see that the Angular compiler is complaining that the app-search-form isn’t found. Why is this? Well, the reason is when we originally created the ClientPage component in [Chapter 5: Components, Templates, and Forms](https://www.educative.io/collection/page/10370001/4603693004488704/6258237853663232), we added the new SearchForm component to the ClientPage but declared the SearchForm component as part of the AppModule.

Now, the ClientPage component became part of the ClientModule. But, we did not make the SearchForm part of this ClientModule. When the ClientPage was loaded into our application, the Angular compiler was also looking for the SearchForm component within the ClientModule because the ClientPage is now part of this module. But as we haven’t moved the SearchForm into the ClientModule, the Angular compiler can’t find the SearchForm in this ClientModule, so it throws an error.

So how can we fix this? Well, there are a number of ways. We could:

* Add the SearchForm component to the ClientModule.
* Move the SearchForm component out of the ClientPage component’s template and add it to the AppComponent template.

Both of these options are great, but let’s think about the role of the SearchForm component for a second to decide what is the better solution.

## Create a shared module

The SearchForm component will allow the user to search through either the Client’s list or the Company’s list to find a matching record. So the SearchForm component will be used in either the **Client** section or the **Company** section, and its functionality will be shared across both areas of the application. This sounds like the SearchForm component needs to be part of a SharedModule, which is one of the modules we listed as part of our list of modules for the application.

### Step 1: Create shared.module.ts file

So let’s create this new shared module and add the SearchForm to it. First, we need to create a new folder for this shared module. This folder can also later house other components that will be shared across both sections. Under the app folder, create a new folder called shared and in this folder, create a new TypeScript file called shared.module.ts, and in that new TypeScript file, we add this code to make our new ShareModule as follows:

import { NgModule } from "@angular/core";  
import { CommonModule } from "@angular/common";  
@NgModule({  
  imports: [CommonModule],  
})  
export class SharedModule {}

Now, in order to get the SearchForm as part of this new SharedModule, we need to add SearchForm component to the export array of the SharedModule, as shown here:

@NgModule({  
  declarations: [SearchFormComponent],  
  imports: [CommonModule, FormsModule],  
  exports: [SearchFormComponent],  
})

You may also notice we’ve added the FormsModule, The SearchForm component needs to have a reference to this module because we’re using a form in the SearchForm component.

📝 **Note:** We will remove SearchFormComponent from the declarations array of app.module.ts file.

### Step 2: Update app.module.ts

Once we’ve created this module, we simply add it to the list of modules in the AppModule imports array like so:

imports: [  
  BrowserModule,  
  AppRoutingModule,  
  BrowserAnimationsModule,  
  ClientModule,  
  CompanyModule,  
  SharedModule,  
  FormsModule,  
  ReactiveFormsModule,  
];

### Step 3: Update client.module.ts

With this new SharedModule, we can add this to the ClientModule imports array in order to make the SearchForm available to the ClientPage component if we update our ClientModule to import the SharedModule as follows:

@NgModule({  
  declarations: [ClientPageComponent, ClientFormComponent],  
  imports: [CommonModule, ReactiveFormsModule, SharedModule],  
})  
  
export class ClientModule {}

## Shared module in action

If we rerun the application, we should see the SearchForm component in the Client page as below:

📝 **Note:** Press the **RUN** button to compile and serve the application. Once the app compiles, you can see its output in the **Output** tab or by clicking on the URL given after Your app can be found at.

###### /

e2e

src

app

shared

shared.module.ts

company

clients

search

app-routing.module.ts

app.component.html

app.component.scss

app.component.spec.ts

app.component.ts

app.module.ts

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.scss

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**shared.module.ts**

1

2

3

4

5

6

7

8

9

10

11

12

import { NgModule } from "@angular/core";

import { CommonModule } from "@angular/common";

import { SearchFormComponent } from '../search/search-form/search-form.component';

import { FormsModule } from '@angular/forms';

@NgModule({

  declarations: [SearchFormComponent],

  imports: [CommonModule, FormsModule],

  exports: [SearchFormComponent],

})

export class SharedModule {}





Run

Save

Reset

**Your app can be found at:**[https://ed-6434592358924288.educative.run](https://ed-6434592358924288.educative.run/)

Client Contacts Manager application after adding sharedModule

We have made a lot of changes to our application, so before we move on, let’s recap what we’ve done:

* Created a new module for the Client section and the Company section
* Added these new modules to AppModule
* Moved all the Client components to the ClientModule
* Created a new SharedModule
* Added the SearchForm to the SharedModule
* Exported the SearchForm from the SharedModule so it can be found through the SharedModule
* Added the SharedModule to both the AppModule and ClientModule so that these modules and their components can access the SearchForm

So, we have basically taken our application and modularised it. As we move forward, adding more functionality to the application and the architecture of the app remains separated and easy to understand.

# Create CustomMaterialModule for Application

Let's explore why we need 'CustomMaterialModule' to import modules from the Angular Material library and see some of the Angular Material components we will use in our application.

**We'll cover the following**

* [Why is CustomMaterialModule needed?](https://www.educative.io/courses/getting-started-with-angular/N8x06qvVGpv#Why-is-CustomMaterialModule-needed?)
  + [Solution 1: Import AngularMaterial modules within our AppModule](https://www.educative.io/courses/getting-started-with-angular/N8x06qvVGpv#Solution-1:-Import-AngularMaterial-modules-within-our-AppModule)
  + [Solution 2: Create a wrapper module](https://www.educative.io/courses/getting-started-with-angular/N8x06qvVGpv#Solution-2:-Create-a-wrapper-module)
* [Create CustomMaterialModule](https://www.educative.io/courses/getting-started-with-angular/N8x06qvVGpv#Create-CustomMaterialModule)
* [Update app.module.ts](https://www.educative.io/courses/getting-started-with-angular/N8x06qvVGpv#Update-app.module.ts)
* [Categories of Angular Material Components](https://www.educative.io/courses/getting-started-with-angular/N8x06qvVGpv#Categories-of-Angular-Material-Components)

Now, while our application is modularized and well structured, it doesn’t look that good. We’re going to use Angular Material to make the application look far more polished and presentable than it is now.

In order to add Angular Material, we need to start adding in the Modules from the Angular Material library that we added back in the [Getting Started with the Angular CLI chapter](https://www.educative.io/collection/page/10370001/4603693004488704/4614882778415104). By doing this, we will see how the use of Modules can quickly add a completely new design to our application.

## Why is CustomMaterialModule needed?

Earlier, when we broke down the list of modules for this application, one of the modules we discussed was the CustomMaterialModule. This module allows us to **add modules from the Angular Material library into one module of our application**. Then, we will make this custom module accessible throughout our application.

### Solution 1: Import AngularMaterial modules within our AppModule

The CustomMaterialModule has a unique use case. The reason we’re creating this module is that the Angular Material UI library has a large set of UI components. In order to make the UI library as small as possible, the team who run the Angular Material project have set up the library so that, in order to use a certain UI component, we need to import the *NgModule* of that component.

For example, we are going to add a new toolbar to the top of our application. To do this, we need to import into our application the MatToolbarModule from Angular Material. Now, we could do this import within our AppModule imports array. But that isn’t a good idea because it will lead to the AppModule having a large imports array and make it hard to read the AppModule code to know what is happening.

### Solution 2: Create a wrapper module

So a way around this issue is to create a wrapper module where we can import in all the modules from Angular Material we want, then we simply **import this wrapper module into our main AppModule**. This wrapper module, CustomMaterialModule's sole job is to import the modules we need from Angular Material, so the code within the module is clean. It’s just the imports array. Then adding this to the AppModule helps keep the **AppModule code clean and readable.**.

📝 **Note:** This idea of having a wrapper for the Angular Material components is something that the [official documentation](https://v5.material.angular.io/guide/getting-started#step-3-import-the-component-modules) of Angular Material Material suggests.

This is why we will create the CustomMaterialModule in order to add all these Angular Material modules in one wrapper module.

It is better practice to add all the UI modules we need from the Angular Material library into one custom module and then share it with our own modules. This is similar to what we have just done with the SharedModule, where we added the SearchForm to this SharedModule and imported that module to the ClientModule in order to make the SearchForm component accessible to the ClientPage component. This time though, instead of sharing our own components, we are sharing the AngularMaterial components throughout our application.

## Create CustomMaterialModule

The first thing we need to do is create this new **CustomMaterialModule**. To do this, we need to create a new TypeScript file in the app folder and call it custom-material.module.ts. Then, in that file add the following code:

import { NgModule } from "@angular/core";  
@NgModule({})  
export class CustomMaterialModule {}

## Update app.module.ts

Next, as we did with the SharedModule, we need to import this new module into the imports array of AppModule as follows:

###### /

e2e

src

app

custom-material.module.ts

shared

company

clients

search

app-routing.module.ts

app.component.html

app.component.scss

app.component.spec.ts

app.component.ts

app.module.ts

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.scss

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**app.module.ts**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

import { BrowserModule } from '@angular/platform-browser';

import { NgModule } from '@angular/core';

import { AppRoutingModule } from './app-routing.module';

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

import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { ClientModule } from './clients/client.module';

import { CompanyModule } from './company/company.module';

import { SharedModule } from './shared/shared.module';

import { CustomMaterialModule } from './custom-material.module';

@NgModule({

  declarations: [

    AppComponent,

  ],

  imports: [

    BrowserModule,

    AppRoutingModule,

    BrowserAnimationsModule,

    ClientModule,

    CompanyModule,

    SharedModule,

    CustomMaterialModule,

  ],

  providers: [],

  bootstrap: [AppComponent]

})

export class AppModule { }





Run

Save

Reset

**Your app can be found at:**[https://ed-6434592358924288.educative.run](https://ed-6434592358924288.educative.run/)

Client Contacts Manager application after CustomerMaterialModule

## Categories of Angular Material Components

Now we can start adding Material modules to this CustomMaterialModule. But first, let’s have a look at Angular Material and the components we might want to add.

If we go to the [Angular Material website](https://material.angular.io/), we can see there are a number of components we can add to our application.

From this list, we can see there are a large number of components we could use in our application, but as the application currently stands, we only need to **add some layout components and form components**. Let’s improve the layout using the Angular Material Components in the [next lesson](https://www.educative.io/collection/page/10370001/4603693004488704/6321226841587712).

# Adding Angular Material Modules to Application

Let's use Angular Material UI components to make the application look far more polished and presentable.

**We'll cover the following**

* [Amending the layout](https://www.educative.io/courses/getting-started-with-angular/xoVZwgvpJr3#Amending-the-layout)
* [Application with Material UI in action](https://www.educative.io/courses/getting-started-with-angular/xoVZwgvpJr3#Application-with-Material-UI-in-action)
  + [Step 1: Update CustomMaterialModule](https://www.educative.io/courses/getting-started-with-angular/xoVZwgvpJr3#Step-1:-Update-CustomMaterialModule)
  + [Step 2: Update AppComponent](https://www.educative.io/courses/getting-started-with-angular/xoVZwgvpJr3#Step-2:-Update-AppComponent)
  + [Step 3: Update SearchFormComponent](https://www.educative.io/courses/getting-started-with-angular/xoVZwgvpJr3#Step-3:-Update-SearchFormComponent)
  + [Step 4: Update search-form.component.scss](https://www.educative.io/courses/getting-started-with-angular/xoVZwgvpJr3#Step-4:-Update-search-form.component.scss)
  + [Step 5: Update ClientFormComponent](https://www.educative.io/courses/getting-started-with-angular/xoVZwgvpJr3#Step-5:-Update-ClientFormComponent)
  + [Step 6: Update client-form.component.scss](https://www.educative.io/courses/getting-started-with-angular/xoVZwgvpJr3#Step-6:-Update-client-form.component.scss)
  + [Step 7: Update import array](https://www.educative.io/courses/getting-started-with-angular/xoVZwgvpJr3#Step-7:-Update-import-array)
  + [Step 8: Update style.scss](https://www.educative.io/courses/getting-started-with-angular/xoVZwgvpJr3#Step-8:-Update-style.scss)
* [Final verdict](https://www.educative.io/courses/getting-started-with-angular/xoVZwgvpJr3#Final-verdict)

## Amending the layout

In the [previous lessons](https://www.educative.io/collection/page/10370001/4603693004488704/4975485825056768), we have seen that our application looks like this:

Client Contacts Manager Application with simple layout

First, let’s tackle the layout of our application. We’re going to:

* Add a toolbar across the top of the app.
* Put the Client form and Search forms in Card components to give them a container to sit-in.
* Update all the form controls to use Angular Material forms, giving them a cleaner look.
* Add one of the predefined themes of Angular Material, making the app look fresher.

## Application with Material UI in action

This is how our application will look like after adding Material UI components.

📝 **Note:** Press the **RUN** button to compile and serve the application. Once the app compiles, you can see its output in the **Output** tab or by clicking on the URL given after Your app can be found at.

###### /

e2e

src

app

custom-material.module.ts

shared

shared.module.ts

company

clients

search

app-routing.module.ts

app.component.html

app.component.scss

app.component.spec.ts

app.component.ts

app.module.ts

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.scss

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**shared.module.ts**

1

2

3

4

5

6

7

8

9

10

11

12

13

import { NgModule } from "@angular/core";

import { CommonModule } from "@angular/common";

import { SearchFormComponent } from '../search/search-form/search-form.component';

import { FormsModule } from '@angular/forms';

import { CustomMaterialModule } from '../custom-material.module';

@NgModule({

  declarations: [SearchFormComponent],

  imports: [CommonModule, FormsModule, CustomMaterialModule],

  exports: [SearchFormComponent],

})

export class SharedModule {}





Run

Save

Reset

**Your app can be found at:**[https://ed-6434592358924288.educative.run](https://ed-6434592358924288.educative.run/)

Client Contacts Manager application with Material UI

You will see the following in the output:

Client Contacts Manager Application with cleaner look

This looks far cleaner than the previous version. Let’s go through the code changes we’ve made in order to achieve this look.

### Step 1: Update CustomMaterialModule

In the custom-material.module.ts, we have added all the modules from Angular Material we needed to achieve this look. Now our CustomMaterialModule looks like this:

import { NgModule } from "@angular/core";  
import { MatToolbarModule } from "@angular/material/toolbar";  
import { MatCardModule } from "@angular/material/card";  
import { MatFormFieldModule } from "@angular/material/form-field";  
import { MatInputModule } from "@angular/material/input";  
import { MatButtonModule } from "@angular/material/button";  
@NgModule({  
  imports: [  
    MatToolbarModule,  
    MatCardModule,  
    MatFormFieldModule,  
    MatInputModule,  
    MatButtonModule,  
  ],  
  exports: [  
    MatToolbarModule,  
    MatCardModule,  
    MatFormFieldModule,  
    MatInputModule,  
    MatButtonModule,  
  ],  
})  
export class CustomMaterialModule {}

Here, we’ve imported and exported all the modules from Angular Material that contain components we are going to use—things like **Material Toolbar** and **Material Card**.

### Step 2: Update AppComponent

Then, we amended the app.component.html template to use the new Toolbar component as follows:

<mat-toolbar> {{ title }} </mat-toolbar>  
<div class="container"><router-outlet></router-outlet></div>

This gives us the dark bar across the top with the title of the application.

### Step 3: Update SearchFormComponent

Next, let’s look at the changes to search-form.component.html as shown here:

<mat-card class="search">  
  <form class="search-form" (ngSubmit)="onSubmit()">  
    <mat-form-field>  
      <input matInput type="text" id="searchBox" placeholder="Search...." name="searchField" [(ngModel)]="searchField" />  
    </mat-form-field>  
    <button mat-raised-button color="primary" type="submit">Submit</button>  
  </form>  
</mat-card>

Here we’ve added the new mat-card component to the template and put the search-form within the Card. Then we added a new mat-form-field around the input form element. The matInput directive tells Angular Material that the input element is going to use the Material look. Finally, we added a mat-raised-button directive to make the Submit button have the Material raised button look.

### Step 4: Update search-form.component.scss

We have also made some changes to the search-form.component.scss to improve the layout as follows:

.search {  
  margin: 5px;  
  
  .search-form {  
    width: 100%;  
    mat-form-field {  
      width: 80%;  
      padding: 15px;  
    }  
  }  
}

### Step 5: Update ClientFormComponent

Moving onto client-form.component.html, we are going to convert the Reactive Form we have in this template to use Material UI components like so:

<mat-card class="client-form">  
  <mat-card-header>  
    <mat-card-title>Add Client</mat-card-title>  
  </mat-card-header>  
  <form [formGroup]="clientForm" (ngSubmit)="saveClient()">  
    <mat-form-field>  
      <input matInput type="text" name="firstnameTxt" formControlName="firstname" placeholder="Firstname..." required />  
    </mat-form-field>  
    <mat-form-field>  
      <input matInput type="text" name="lastnameTxt" formControlName="lastname" placeholder="Lastname..." required />  
    </mat-form-field>  
    <mat-form-field>  
      <input matInput type="email" name="emailTxt" formControlName="email" placeholder="Email..." required />  
    </mat-form-field>  
    <mat-form-field>  
      <input matInput type="tel" name="telephoneTxt" formControlName="telephoneNumber" placeholder="Telephone..." />  
    </mat-form-field>  
    <mat-form-field>  
      <input matInput type="tel" name="companyTxt" formControlName="companyName" placeholder="Company Name..." required />  
    </mat-form-field>  
    <p>  
      <button mat-raised-button color="primary" type="submit" [disabled]="!clientForm.valid">Save</button>  
    </p>  
  </form>  
</mat-card>

In this template, we’ve replaced all the input elements to use the mat-form-field components and added a mat-raised-button directive to the Submit button.

### Step 6: Update client-form.component.scss

We have also made some changes to the client-form.component.scss to improve the layout as follows:

.client-form {  
    padding-top: 20px;  
    margin-top: 40px;  
    mat-form-field {  
        width: 60%;  
    }  
    button {  
        font: 1em sans-serif;  
    }  
}

We’ve added the width to the mat-form-field in order to make all the form fields the same width.

### Step 7: Update import array

Then, you have to add CustomMaterialModule in the imports array of shared.module.ts and client.module.ts.

### Step 8: Update style.scss

Finally, the last change we made was to add the Angular Material theme CSS file to the style.scss file as follows:

@import '@angular/material/prebuilt-themes/pink-bluegrey.css';  
 html,  
body {  
    height: 100%;  
}  
body {  
    margin: 0;  
    font-family: Roboto, 'Helvetica Neue', sans-serif;  
}

## Final verdict

As you can see, by importing the module from Angular Material, we’ve been able to add just a couple of extra components into our application that drastically change the UI. This shows how modules in Angular can add large amounts of functionality (for example, a completely new UI) to an application using only a couple of lines of code.

We can make use of third-party modules to plugin functionality to extend our applications. This ability to add new modules of functionality to our Angular applications is a real benefit. If we need to add a new feature to an application and there is a third-party module that helps with this feature, we can simply plug in this third-party module and start using the functionality it provides.

# Routing and Navigation

Let's learn about routing and navigation in this chapter.

**We'll cover the following**

* [Chapter learning outcomes](https://www.educative.io/courses/getting-started-with-angular/3jm34D8lkAr#Chapter-learning-outcomes)
* [Client contacts manager application](https://www.educative.io/courses/getting-started-with-angular/3jm34D8lkAr#Client-contacts-manager-application)

## Chapter learning outcomes

So far, we’ve looked at what Angular is, how to get started, and how to use the Angular CLI to help us build our first Angular application. However, the application we’ve built so far doesn’t have a lot of functionality. We will start adding more functionality to our application over the next couple of chapters in order to make it more interactive.

In this chapter, we will start by looking at **Routing and Navigation in Angular** and how we can use the features that Angular provides for routing to **create complex and sophisticated navigation systems** to support our application’s needs.

In this chapter, we will cover the following topics:

* What are the routes?
* How routes work in Angular
* How to set up routes for your Angular application
* How to add navigation to an Angular application
* What route guards are and how to use them
* What asynchronous routing is and what benefits it can bring to your Angular application

## Client contacts manager application

We will be expanding on our Client Contacts Manager application by adding navigation to it so that a user can navigate between the sections of the application.

📝 **Note:** Here is the preview of what we will create in this chapter. Press the **RUN** button to compile and serve the application. Once the app compiles, you can see its output in the **Output** tab or by clicking on the URL given after Your app can be found at.

###### /

e2e

src

app

about

navigation

page-not-found

custom-material.module.ts

shared

company

clients

client.module.ts

client-form

client-page

client-page.component.html

client-page.component.scss

client-page.component.spec.ts

client-page.component.ts

search

search-form

app-routing.module.ts

app.component.html

app.component.scss

app.component.spec.ts

app.component.ts

app.module.ts

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.scss

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**client-page.component.tsclient-page.component.spec.ts**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

import { ComponentFixture, TestBed } from '@angular/core/testing';

import { ClientPageComponent } from './client-page.component';

describe('ClientPageComponent', () => {

  let component: ClientPageComponent;

  let fixture: ComponentFixture<ClientPageComponent>;

  beforeEach(async () => {

    await TestBed.configureTestingModule({

      declarations: [ ClientPageComponent ]

    })

    .compileComponents();

  });

  beforeEach(() => {

    fixture = TestBed.createComponent(ClientPageComponent);

    component = fixture.componentInstance;

    fixture.detectChanges();

  });

  it('should create', () => {

    expect(component).toBeTruthy();

  });

});





Run

Save

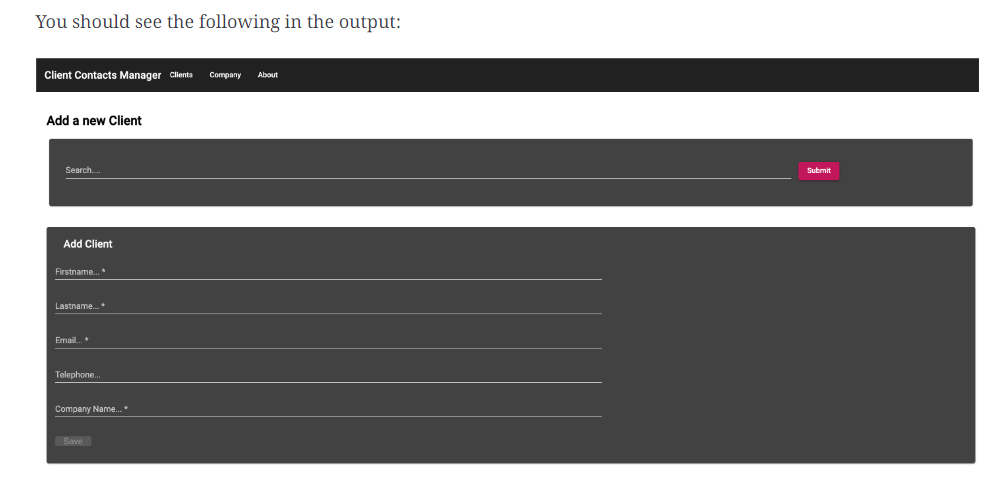
Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/>

Client Contacts Manager Application with different routes

You should see the following in the output:

Client Contacts Manager application with navigation





# What Are Routes?

Familiarize yourself with the basic routing module and how different route patterns are registered within our application.

**We'll cover the following**

* [What are the routes?](https://www.educative.io/courses/getting-started-with-angular/g2KxQQQJY93#What-are-the-routes?)
* [How does the browser support navigation?](https://www.educative.io/courses/getting-started-with-angular/g2KxQQQJY93#How-does-the-browser-support-navigation?)
* [How does the router support routing?](https://www.educative.io/courses/getting-started-with-angular/g2KxQQQJY93#How-does-the-router-support-routing?)
* [Routing module for client contacts manager application](https://www.educative.io/courses/getting-started-with-angular/g2KxQQQJY93#Routing-module-for-client-contacts-manager-application)

## What are the routes?

A **route** is when the user navigates from one section to the next.

We call them routes because all the navigation in an Angular application is set up and managed through the Angular Router. This ***Angular Router*** is another module that allows us as Angular developers to set up the navigation within our application.

Route

## How does the browser support navigation?

There are various ways in which the browser supports navigation:

* When the URL in the browser’s address bar is changed, and the browser navigates to the new section.
* When the user clicks on a link in a page of the application, forcing the app to navigate to a new page.
* When the user clicks on either the back or forward buttons of the browser, navigating through the local browser history.

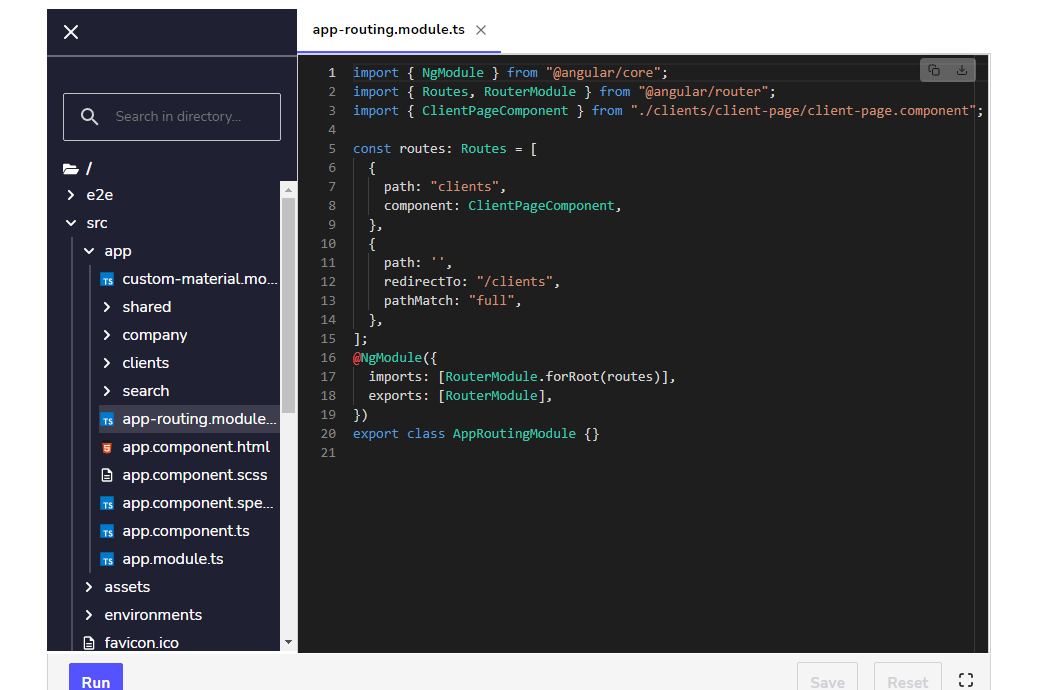
## How does the router support routing?

When the URL of the browser changes, the router can take that new URL as instruction and navigate to the relevant section of the application.

We can add links within our views so that the user can click on these links to navigate to different parts of the application. The router has been designed to keep a log of the activity in the browser’s history, so support for the back and forward buttons are available in the Angular Router.

## Routing module for client contacts manager application

When we originally started our Client Contacts Manager application, the Angular CLI asked us whether we wanted to set up routing in our application, to which we said “yes”. Therefore, the CLI created a very basic routing module in a TypeScript file called app-routing.module.ts. If we open that file, we can see how routes are set up:



Routing module of Clients Contact Manager application

There isn’t a lot going on here at the moment – the important part is seeing how Routes are defined and what modules are involved in building the routes of the application.

As you can see, there is an array called routes, and this array is passed onto the forRoot() function of RouterModule (*line 17*). This RouterModule is then exported out of our AppRoutingModule. AppRoutingModule is then imported into the main AppModule:

imports: [  
    BrowserModule,  
    AppRoutingModule,  
    BrowserAnimationsModule,  
    ClientModule,  
    CompanyModule,  
    SharedModule,  
    CustomMaterialModule,  
  ],

That’s all great, but what does this actually mean? Well, in our AppRoutingModule, we create an array of objects that have a type called Route. These objects have properties that are used to create a model of all the possible routes the application needs to support. This array is passed to the Angular framework’s RouterModule so that the framework registers the patterns we’ve set up in the array.

# The Routes Array

Let's explore how the routes are set up using the Routes array.

**We'll cover the following**

* [What is the Routes array?](https://www.educative.io/courses/getting-started-with-angular/q2NBNgJynzD#What-is-the-Routes-array?)
* [The interface of the Route object](https://www.educative.io/courses/getting-started-with-angular/q2NBNgJynzD#The-interface-of-the-Route-object)
* [Route with /clients path property](https://www.educative.io/courses/getting-started-with-angular/q2NBNgJynzD#Route-with-/clients-path-property)
* [Create a Route with /company path property](https://www.educative.io/courses/getting-started-with-angular/q2NBNgJynzD#Create-a-Route-with-/company-path-property)
  + [Step 1: Generate CompanyPageComponent](https://www.educative.io/courses/getting-started-with-angular/q2NBNgJynzD#Step-1:-Generate--CompanyPageComponent)
  + [Step 2: Update company.module.ts](https://www.educative.io/courses/getting-started-with-angular/q2NBNgJynzD#Step-2:-Update-company.module.ts)
  + [Step 3: Update routes array](https://www.educative.io/courses/getting-started-with-angular/q2NBNgJynzD#Step-3:-Update-routes-array)
* [Application with company path in action](https://www.educative.io/courses/getting-started-with-angular/q2NBNgJynzD#Application-with-company-path-in-action)

## What is the Routes array?

The **Route array** contains a set of objects that define the details of the Route.

## The interface of the Route object

If we quickly look at the interface that defines what Route object properties are, we can see that there are a number of properties available to us:

interface Route {  
       path?: string  
       pathMatch?: string  
       matcher?: UrlMatcher  
       component?: Type<any>  
       redirectTo?: string  
       outlet?: string  
       canActivate?: any[]  
       canActivateChild?: any[]  
       canDeactivate?: any[]  
       canLoad?: any[]  
       data?: Data  
       resolve?: ResolveData  
       children?: Routes  
       loadChildren?: LoadChildren  
       runGuardsAndResolvers?: RunGuardsAndResolvers  
}

📝 **Note:** This is the interface that defines a Route in Angular.

As you can see, there are a lot of different properties we can use to create a route array. This array of Route objects makes up a model of how the application will handle different paths into the application.

## Route with /clients path property

For example, in AppRoutingModule, we have a Route that has two properties: one called path and another called component:

const routes: Routes = [  
  {  
    path: "clients",  
    component: ClientPageComponent,  
  },  
];

When the browser loads a URL with clients after the domain name, for example, http://localhost:4200/clients, the clients part of the URL matches the path property. The router, then, knows to read from the other properties of the matching object. The other property of this object is the component property. The Route module takes the value of this component property, which in this case is ClientPageComponent, and it loads it into the main view of the component that has the same name.

Working flow of '/client' path

We can add to this Routes array and add other route objects with path properties so that they can match other paths that our application may need.

## Create a Route with /company path property

Let’s add the company path in our Routes array.

### Step 1: Generate CompanyPageComponent

Let’s build CompanyPageComponent under the company folder. First, navigate to the company folder and then run the following command in the terminal:

ng generate component company-page

📝 **Note:** Press the **Run** button and see what happens!

###### /

e2e

src

app

custom-material.module.ts

shared

company

clients

client.module.ts

client-form

client-page

search

app-routing.module.ts

app.component.html

app.component.scss

app.component.spec.ts

app.component.ts

app.module.ts

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.scss

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**app.module.tsapp-routing.module.tscompany.module.tsclient.module.ts**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

import { NgModule } from "@angular/core";

import { CommonModule } from "@angular/common";

import { ClientPageComponent } from "./client-page/client-page.component";

import { ClientFormComponent } from "./client-form/client-form.component";

import { ReactiveFormsModule } from "@angular/forms";

import { SharedModule } from '../shared/shared.module';

import { CustomMaterialModule } from '../custom-material.module';

@NgModule({

  declarations: [ClientPageComponent, ClientFormComponent],

  imports: [CommonModule, ReactiveFormsModule, SharedModule, CustomMaterialModule],

})

export class ClientModule {}





Run

Save

Reset

**Your app can be found at:**[https://ed-6434592358924288.educative.run](https://ed-6434592358924288.educative.run/)

Create company-page module

Then, something strange will happen. The Angular CLI will give us an error saying , “More than one module matches. Use *skip-import* option to skip importing the component into the closest module.”

This is where the Angular CLI is saying that it doesn’t know what module to add this new component to. Since we now have a few modules, the Angular CLI can’t do what it normally does when creating a component, that is, add it to a modules declaration array.

This is fine. We can still create the component, but we now need to add the component to a module ourselves. In order to create the Navigation component, we need to run the following command:

ng generate component  company-page --skip-import

📝 **Note:** Run this command in the above terminal and see what happens!

We are using the skip-import option for the generate command of the CLI to tell the CLI not to try and automatically import the component for us. Adding this option will allow the CLI to just generate the component.

This will create a new component under the company folder.

### Step 2: Update company.module.ts

Now put the CompanyPageComponent in the declarations array of company module:

@NgModule({  
  imports: [CommonModule],  
  declarations: [CompanyPageComponent]  
})

### Step 3: Update routes array

Now we need to add the import statement to the app-routing.module.ts file:

import{CompanyPageComponent}from'./company/company-page/company-page.component';

Also, let’s add a new object with a path of the company and a component of CompanyPageComponent, like this:

const routes: Routes = [  
  {  
    path: "clients",  
    component: ClientPageComponent,  
  },  
  {  
     path: 'company',  
     component: CompanyPageComponent,  
  },  
  {  
    path: '',  
    redirectTo: "/clients",  
    pathMatch: "full",  
  },  
];

Here, when http://localhost:4200/company is entered into a browser address bar, the company route will match that path, and the CompanyPageComponent will load.

## Application with company path in action

📝 **Note:** Press the **RUN** button to compile and serve the application. Once the app compiles, click on the URL given after Your app can be found at, and see what happens!

###### /

e2e

src

app

custom-material.module.ts

shared

company

clients

search

app-routing.module.ts

app.component.html

app.component.scss

app.component.spec.ts

app.component.ts

app.module.ts

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.scss

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**app-routing.module.ts**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

import { NgModule } from "@angular/core";

import { Routes, RouterModule } from "@angular/router";

import { ClientPageComponent } from "./clients/client-page/client-page.component";

import { CompanyPageComponent } from './company/company-page/company-page.component';

const routes: Routes = [

  {

    path: "clients",

    component: ClientPageComponent,

  },

  {

    path: 'company',

    component: CompanyPageComponent,

  },

  {

    path: '',

    redirectTo: "/clients",

    pathMatch: "full",

  },

];

@NgModule({

  imports: [RouterModule.forRoot(routes)],

  exports: [RouterModule],

})

export class AppRoutingModule {}





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/company>

Routing module of Clients Contact Manager application with company path

You should see the following in the output:

Client Contacts Manager Application with company route

As we saw from the Route interface, there are more than just path and component properties available to the Route object, which we will be using as we go further into routing and the navigation of Angular. But, from the current AppRoutingModule, we can see how routes are set up.

# The Router Outlet Component

Learn what the Router Outlet component is, where the component is loaded and how a user can see the related component of a route.

**We'll cover the following**

* [What is the Router Outlet component?](https://www.educative.io/courses/getting-started-with-angular/gkRkmV4xJO6#What-is-the-Router-Outlet-component?)
* [Router Outlet directive in AppComponent template](https://www.educative.io/courses/getting-started-with-angular/gkRkmV4xJO6#Router-Outlet-directive-in-AppComponent-template)
* [What does the router-outlet directive do?](https://www.educative.io/courses/getting-started-with-angular/gkRkmV4xJO6#What-does-the-router-outlet-directive-do?)

## What is the Router Outlet component?

The **Router Outlet** is a directive that’s part of the Router module. Its main role is to be a placeholder within our application where the Angular framework should place any components within a template.

## Router Outlet directive in AppComponent template

Currently, we are using this directive in the app.component.html file of our Client Contacts Manager application. If we open this file, we will see the following:

###### /

e2e

src

app

custom-material.module.ts

shared

company

clients

search

app-routing.module.ts

app.component.html

app.component.scss

app.component.spec.ts

app.component.ts

app.module.ts

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.scss

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**app.component.html**

1

2

3

4

5

<mat-toolbar> {{ title }} </mat-toolbar>

<div class="container">

    <router-outlet></router-outlet>

</div>





Run

Save

Reset

**Your app can be found at:**[https://ed-6434592358924288.educative.run](https://ed-6434592358924288.educative.run/)

Router Outlet directive used in app.component.html

Here, we can see that the <router-outlet></router-outlet> selector is placed in the containing div, underneath the new Material Toolbar component we added in [NgModules chapter](https://www.educative.io/collection/page/10370001/4603693004488704/6321226841587712" \t "_blank).

## What does the router-outlet directive do?

So, what does this directive do? Well, in the Routes array, we specify what path the RouterModule needs to be aware of, and when it matches one of these routes, it needs to load a component, which is set as the component property of the Route object we’re creating.

const routes: Routes = [  
  {  
    path: "clients",  
    component: ClientPageComponent,  
  },  
  {  
    path: 'company',  
    component: CompanyPageComponent,  
  },  
  {  
    path: '',  
    redirectTo: "/clients",  
    pathMatch: "full",  
  },  
];

When Angular sees that path, it knows what component to load and uses the placeholder Router-Outlet directive as the place in our template to load the template of the specified component.

💡 **Point to ponder**

Remove the <router-outlet></router-outlet> selector from the above app and see what happens!

In our app.component.html template, we’ve put the Router-Outlet selector in a div underneath the toolbar. This shows that it can be part of a complete template. It doesn’t need to be in a template on its own. If you want to have a header and footer, which is always displayed to the user as they go from section to section in your application, you would add them to the main app.component.html template and then put the <router-outlet></router-outlet> selector between the header and footers so that when a new route loads, the header and footer are always displayed.

Router-Outlet directive

The Router-Outlet directive, while it looks simple, is a powerful and important part of how navigation works within Angular.

# Wildcard Routes

Learn what wildcard routes are and how we can use them to handle invalid URLs within our application.

**We'll cover the following**

* [What are the wildcard routes?](https://www.educative.io/courses/getting-started-with-angular/39nLJk7X27Q#What-are-the-wildcard-routes?)
  + [Example: Navigating an invalid route](https://www.educative.io/courses/getting-started-with-angular/39nLJk7X27Q#Example:-Navigating--an-invalid-route)
* [Handling invalid routes](https://www.educative.io/courses/getting-started-with-angular/39nLJk7X27Q#Handling-invalid-routes)
  + [Step 1: Set path as \*\*](https://www.educative.io/courses/getting-started-with-angular/39nLJk7X27Q#Step-1:-Set-path-as-**)
  + [Step 2: Generate component](https://www.educative.io/courses/getting-started-with-angular/39nLJk7X27Q#Step-2:-Generate--component)
  + [Step 3: Update template](https://www.educative.io/courses/getting-started-with-angular/39nLJk7X27Q#Step-3:-Update-template)
  + [The wildcard route in action](https://www.educative.io/courses/getting-started-with-angular/39nLJk7X27Q#The-wildcard-route-in-action)

## What are the wildcard routes?

Another important part of defining Routes in our application is setting up wildcard routes.

**Wildcard routes** are how we tell Angular how to handle invalid URLs and what to do with them.

An invalid URL could be where the user mistypes the name of a path.

### Example: Navigating an invalid route

For example, we know that http://localhost:4200/clients is a valid path in our Client Contact Manager application, but http://localhost:4200/client isn’t because we don’t have a route in our Routes array that matches that exact /client route. If the user was to add that into the browser’s address bar, our application would throw an error saying that this path is not recognized.

📝 **Note:** Press the **RUN** button to compile and serve the application. Once the app compiles, click on the URL given after Your app can be found at, and see what happens!

###### /

e2e

src

app

custom-material.module.ts

shared

company

clients

search

app-routing.module.ts

app.component.html

app.component.scss

app.component.spec.ts

app.component.ts

app.module.ts

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.scss

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**app-routing.module.ts**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

import { NgModule } from "@angular/core";

import { Routes, RouterModule } from "@angular/router";

import { ClientPageComponent } from "./clients/client-page/client-page.component";

import { CompanyPageComponent } from './company/company-page/company-page.component';

const routes: Routes = [

  {

    path: "clients",

    component: ClientPageComponent,

  },

  {

     path: 'company',

     component: CompanyPageComponent,

  },

  {

    path: '',

    redirectTo: "/clients",

    pathMatch: "full",

  },

];

@NgModule({

  imports: [RouterModule.forRoot(routes)],

  exports: [RouterModule],

})

export class AppRoutingModule {}





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/client>

Navigation to invalid URL

If you open the console in your browser, you will see the following error:

Error: Cannot match any routes

So, does this mean that we need to add routes for every eventuality? Thankfully, no – the Route module has a solution to this problem, wildcard routes.

## Handling invalid routes

A wildcard route has a path set as \*\*, which will then match all routes into the application. This doesn’t mean that it will run every time a path is added to the browser address bar. What happens is the Angular route takes the URL from the address bar, runs it through the list of Routes we define in the Routes array, and goes through all the other routes. If one of them matches the URL, it runs it. If the URL doesn’t match any of the standard routes, it runs the wildcard route.

### Step 1: Set path as \*\*

This is what a wildcard route will look like in the routes array of our Client Contacts Manager application:

const routes: Routes = [  
  {  
    path: "clients",  
    component: ClientPageComponent,  
  },  
  {  
     path: 'company',  
     component: CompanyPageComponent,  
  },  
  {  
    path: '',  
    redirectTo: "/clients",  
    pathMatch: "full",  
  },  
  {  
    path: '\*\*',  
    component: PageNotFoundComponent,  
  },  
];

As you can see, we’ve added a different component, PageNotFoundComponent. This can be a simple component just displays a message to the user stating that the URL they have entered can’t be found. We could show them an image to highlight what’s gone wrong, and we could also add a link in the PageNotFoundComponent component back to the /clients section, which we know is a valid URL of our application.

Working flow of wildcard route

### Step 2: Generate component

Let’s build this PageNotFoundComponent. All we need to do is use the following command:

ng generate component page-not-found

📝 **Note:** Click the terminal window and see what happens!

This code requires the following environment variables to execute:

NG\_CLI\_ANALYTICS

off

npm\_config\_loglevel

silent

CHROME\_BIN

usr/bin/google-chrome

Edit

**Terminal 1**

Terminal

**Click to Connect...**

****

This will create a new component under the app folder.

### Step 3: Update template

In the template of this folder, we can add the following code:

<h2>Sorry, the page you are looking for has not been found</h2>

### The wildcard route in action

Now, you will see the following message whenever you navigate to an invalid router:

Sorry, the page you are looking for has not been found

📝 **Note:** Press the **RUN** button to compile and serve the application. Once the app compiles, click on the URL given after Your app can be found at, and see what happens!

###### /

e2e

src

app

page-not-found

custom-material.module.ts

shared

company

clients

search

app-routing.module.ts

app.component.html

app.component.scss

app.component.spec.ts

app.component.ts

app.module.ts

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.scss

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**app-routing.module.ts**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

import { NgModule } from "@angular/core";

import { Routes, RouterModule } from "@angular/router";

import { ClientPageComponent } from "./clients/client-page/client-page.component";

import { CompanyPageComponent } from './company/company-page/company-page.component';

import { PageNotFoundComponent } from './page-not-found/page-not-found.component';

const routes: Routes = [

  {

    path: "clients",

    component: ClientPageComponent,

  },

  {

    path: 'company',

    component: CompanyPageComponent,

  },

  {

    path: '',

    redirectTo: "/clients",

    pathMatch: "full",

  },

  {

    path: '\*\*',

    component: PageNotFoundComponent,

  },

];

@NgModule({

  imports: [RouterModule.forRoot(routes)],

  exports: [RouterModule],

})

export class AppRoutingModule {}





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/client>

Routing Module with Wildcard

You should see the following in the output:

Client Contacts Manager application with PageNotFoundComponent

When we go through how to add links to our application, we can update this template so that we have a link back to the Clients section. That way our user doesn’t get stuck in the PageNotFound view of the application.

# The redirectTo Route

Let's explore how we can use 'redirectTo' route for handling the edge cases of URLs.

**We'll cover the following**

* [What is the redirectTo route?](https://www.educative.io/courses/getting-started-with-angular/xV35Xy0zLrq#What-is-the-redirectTo-route?)
* [Example of redirectTo route](https://www.educative.io/courses/getting-started-with-angular/xV35Xy0zLrq#Example-of-redirectTo-route)
* [Properties of redirectTo route](https://www.educative.io/courses/getting-started-with-angular/xV35Xy0zLrq#Properties-of-redirectTo-route)
  + [redirectTo property](https://www.educative.io/courses/getting-started-with-angular/xV35Xy0zLrq#redirectTo-property)
  + [pathMatch property](https://www.educative.io/courses/getting-started-with-angular/xV35Xy0zLrq#pathMatch-property)
* [Application of redirectTo routes](https://www.educative.io/courses/getting-started-with-angular/xV35Xy0zLrq#Application-of-redirectTo-routes)

## What is the redirectTo route?

So, we know about the wildcard route handler and how it runs when a URL is entered into the address bar that doesn’t match one of our defined routes, but what if we don’t want the user to go to the page not found section? For example, what if the user enters http://localhost:4200/client instead of http://localhost:4200/clients? It seems a bit unfair to send them to a page not found when they were so close to the correct route. Well, this is where redirectTo routes come in.

A **redirectTo route** is a route that we set up as a way of capturing these edge case URLs that the user may put in.

## Example of redirectTo route

For example, here is a redirectTo path for the /client URL:

📝 **Note:** Press the **RUN** button to compile and serve the application. Once the app compiles, click on the URL given after Your app can be found at, and see what happens!

###### /

e2e

src

app

page-not-found

custom-material.module.ts

shared

company

clients

search

app-routing.module.ts

app.component.html

app.component.scss

app.component.spec.ts

app.component.ts

app.module.ts

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.scss

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**app-routing.module.ts**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

import { NgModule } from "@angular/core";

import { Routes, RouterModule } from "@angular/router";

import { ClientPageComponent } from "./clients/client-page/client-page.component";

import { CompanyPageComponent } from "./company/company-page/company-page.component";

import { PageNotFoundComponent } from "./page-not-found/page-not-found.component";

const routes: Routes = [

  {

    path: "clients",

    component: ClientPageComponent,

  },

  {

    path: "client",

    redirectTo: "/clients",

    pathMatch: "full",

  },

  {

    path: "company",

    component: CompanyPageComponent,

  },

  {

    path: '',

    redirectTo: "/clients",

    pathMatch: "full",

  },

  {

    path: "\*\*",

    component: PageNotFoundComponent,

  },

];

@NgModule({

  imports: [RouterModule.forRoot(routes)],





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/client>

Routing Module with Wildcard

In this example, we’re saying that if the user enters http://localhost:4200/client in the address bar, then redirect them to the real path of http://localhost:4200/clients.

This means that we don’t need to add loads of routes for the many variations of the http://localhost:4200/clients URL or give the user a bad experience if they slightly misspell the URL by missing a letter off the end of a path. It’s better to redirect them than show a page not found error just because they’re missing a letter or two.

Working flow of redirectTo route

## Properties of redirectTo route

The redirectTo route has two properties that separate it from the standard type of route:

* redirectTo
* pathMatch

### redirectTo property

The redirectTo property is where we set the actual path we want to redirect the user to if this route is fired. In the preceding example, you can see that we are redirecting the user to the /clients path if this redirect route is fired.

### pathMatch property

The pathMatch property is interesting. Its role is to set what part of the URL matches before the redirectTo route is triggered.

There are two values for the pathMatch property: **full and prefix**. When pathMatch is set to **full**, Angular will look at the entire part of the remaining URL. When we say the remaining URL, we mean the part of the URL that comes after the domain name. http://localhost:4200 is our domain name, while /clients is the remaining part of the complete URL.

If pathMatch is set to **full**, then Angular will look at the full part of the remaining domain and see if that matches one of its redirectTo defined routes.

The other type of pathMatch is a **prefix**. If this is set, then the Angular router matches the redirectTo route with the beginning of the entered URL. So, if the user entered http://localhost:4200/clientsHome, then the /clients redirectTo route would fire because the beginning of /clientsHome has /clients in it.

## Application of redirectTo routes

redirectTo routes become especially useful when the user enters just the domain name in the URL, but we actually want them to go to a specific route when they first go to our application. This type of redirectTo route is set up like this:

{  
    path: '',  
    redirectTo: '/clients',  
    pathMatch: 'full'  
}

In this route, we are telling the Angular Router that if there is no remaining part of the URL to redirects\ the user to the /clients section. That is why the path property is set to ''.

So, when the user first enters the domain name of our application, that is, http://localhost:4200, we want to actually redirect them to http://localhost:4200/clients, as this is the starting point of our application.

As you can see, through the use of wildcard routes and redirectTo routes, we can set up routes to cover most edge cases. Wildcards are used when the user enters a URL that is not like anything we’ve defined as a route.redirectTo routes are for when the user has entered a URL that is close to what we expect or if they have entered just the domain name, and we want to redirect them to the starting point of our application.

# Creating Our Navigation Component

Let's create a navigation component for our Client Contacts Manager application and move the toolbar to this component.

**We'll cover the following**

* [Angular Materials components](https://www.educative.io/courses/getting-started-with-angular/NENmqnD5L8v#Angular-Materials-components)
  + [Menu component](https://www.educative.io/courses/getting-started-with-angular/NENmqnD5L8v#Menu-component)
  + [Sidenav component](https://www.educative.io/courses/getting-started-with-angular/NENmqnD5L8v#Sidenav-component)
  + [Toolbar component](https://www.educative.io/courses/getting-started-with-angular/NENmqnD5L8v#Toolbar-component)
* [Updating the toolbar](https://www.educative.io/courses/getting-started-with-angular/NENmqnD5L8v#Updating-the-toolbar)
  + [Step 1: Create a new component](https://www.educative.io/courses/getting-started-with-angular/NENmqnD5L8v#Step-1:-Create-a-new-component)
  + [Step 2: Update AppModule](https://www.educative.io/courses/getting-started-with-angular/NENmqnD5L8v#Step-2:-Update-AppModule)
  + [Step 3: Update AppComponent](https://www.educative.io/courses/getting-started-with-angular/NENmqnD5L8v#Step-3:-Update-AppComponent)
  + [Step 4: Update NavigationComponent](https://www.educative.io/courses/getting-started-with-angular/NENmqnD5L8v#Step-4:-Update-NavigationComponent)
  + [Step 5: Update NavigationComponent class](https://www.educative.io/courses/getting-started-with-angular/NENmqnD5L8v#Step-5:-Update-NavigationComponent-class)
  + [Step 6: Update AppComponent](https://www.educative.io/courses/getting-started-with-angular/NENmqnD5L8v#Step-6:-Update-AppComponent)

Now that we know more about Routes, the RouterModule, and how to structure a route, it’s time to put what we’ve learned into practice. In this section, we will add a new component to our Client Contacts Manager application. This will be the navigation for the application. First, though, we need to look at the component Angular Material provides to decide what layout and approach we will use.

## Angular Materials components

If we go to the [Angular Material website](https://material.angular.io/) again, under the **Components** section, you’ll see the different components.

We could use the following three components for our menu, as follows:

Angular Material components

### Menu component

The Menu component gives us a floating menu, which has the links in the menu open below the menu item. You might use this if you have a complex application that has many sections with subsections that you want a user to be able to access.

### Sidenav component

The **Sidenav** gives us a side draw layout where the menu slides out from the side when the user has clicked a menu icon. This is a common navigation system for applications that need to work on both the web and mobile. The [Angular website](https://angular.io/docs) uses this type of navigation.

### Toolbar component

The final type is the toolbar, where a bar is drawn across the top of an application. We are currently using this in the Client Contacts application. It’s where the title of the app is displayed.

It is possible to combine these three components. For example, your application could have a toolbar across the top, then have an icon that opens a **Sidenav**. You could also have a menu item in the toolbar that opens a login/logout icon. To see an example of this type of approach, look at the [Angular docs website](https://angular.io/docs).

Angular docs website

In the preceding screenshot, you can see how the layout of the site is using a Sidenav for all the links to the main sections of the documentation, while the toolbar is used to hold the Angular logo and the Menu links to all the other parts of the Angular Docs website.

For our Client Contacts application, we are going to keep the toolbar we currently have and add some menu items in order to create the links to the sections of our application.

## Updating the toolbar

Now, we’re going to update the toolbar of the application. The toolbar component gives us a place within the application to house the navigation of the application. Here, we add the links to the other sections of the application, which, as we will see, makes use of the Routes to move between sections.

Currently, our toolbar is in the app.component.html template:

<mat-toolbar> {{ title }} </mat-toolbar>  
<div class="container">  
    <router-outlet></router-outlet>  
</div>

While we could leave the toolbar component here, it would be better if we created a new navigation component that not only shows the title in the toolbar but also has our menu items.

📝 **Note:** There are many approaches we could take when looking at how to structure our application. What we are aiming to do here is split functionality into as many reasonable components, as we can in order to make it so that we can see what each component does, as well as its role within the application.

### Step 1: Create a new component

The first thing we need to do is create our new Navigation component. Navigate to the app folder of our application in the Terminal and run the following command:

ng generate component navigation --skip-import

We are using the skip-import option for the generate command of the CLI to tell the CLI not to try and automatically import the component for us. Adding this option will allow the CLI to just generate the component.

### Step 2: Update AppModule

Now that the new Navigation component has been created, we can open app.module.ts and add it to the declarations array:

@NgModule({  
  declarations: [AppComponent, NavigationComponent],  
  imports: [  
    BrowserModule,  
    AppRoutingModule,  
    BrowserAnimationsModule,  
    CustomMaterialModule,  
    ClientModule,  
    CompanyModule,  
    SharedModule,  
  ],  
  providers: [],  
  bootstrap: [AppComponent],  
})  
export class AppModule {}

### Step 3: Update AppComponent

Next, we need to update the app.component.html file so that we can use our new Navigation component:

📝 **Note:** Press the **RUN** button to compile and serve the application. Once the app compiles, click on the URL given after Your app can be found at, and see what happens!

###### /

e2e

src

app

navigation

page-not-found

custom-material.module.ts

shared

company

clients

search

app-routing.module.ts

app.component.html

app.component.scss

app.component.spec.ts

app.component.ts

app.module.ts

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.scss

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**app.component.html**

1

2

3

4

5

6

<mat-toolbar> {{ title }} </mat-toolbar>

<app-navigation></app-navigation>

<div class="container">

    <router-outlet></router-outlet>

</div>





Run

Save

Reset

**Your app can be found at:**[https://ed-6434592358924288.educative.run](https://ed-6434592358924288.educative.run/)

Client Contacts Manager application with Navigation component

If we run the site, you should see the following words under the toolbar:

navigation works!

Next, we need to move the current <mat-toolbar> into this new Navigation component.

### Step 4: Update NavigationComponent

In navigation.component.html file of the above SPA widget, replace all the HTML with <mat-toolbar>{{ title }}</mat-toolbar>.

You’ll see the following error:

Navigation component does not know about property title

The issue is that the Navigation component doesn’t know what the property title is.

### Step 5: Update NavigationComponent class

To fix this, we need to add a new @Input() to our Navigation component in order to pass the **title** into the **component**. (We discussed what @Input() was in the [Components, Templates, and Forms chapter](https://www.educative.io/collection/page/10370001/4603693004488704/6556515983949824))

Let’s add a new property to navigation.component.ts:

export class NavigationComponent implements OnInit {  
  @Input() title: string;  
  
  constructor() {}  
  ngOnInit() {}  
}

### Step 6: Update AppComponent

Then, we can update where the Navigation component is used to pass in a value for the title property. In app.component.html, we need to add our title, Client Contacts Manager, property to the <app-navigation> selector:

<app-navigation title="Client Contacts Manager"></app-navigation>  
<div class="container">  
    <router-outlet></router-outlet>  
</div>

After the Angular CLI has rebuilt and reloaded the page, you should see the title of the Client Contacts Manager back in the toolbar.

###### /

e2e

src

app

navigation

navigation.component.ts

navigation.component.spec.ts

navigation.component.scss

navigation.component.html

page-not-found

custom-material.module.ts

shared

company

clients

search

app-routing.module.ts

app.component.html

app.component.scss

app.component.spec.ts

app.component.ts

app.module.ts

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.scss

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**navigation.component.ts**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

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

@Component({

  selector: 'app-navigation',

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

  styleUrls: ['./navigation.component.scss']

})

export class NavigationComponent implements OnInit {

  @Input() title: string;

  constructor() {}

  ngOnInit() {}

}





Run

Save

Reset

**Your app can be found at:**[https://ed-6434592358924288.educative.run](https://ed-6434592358924288.educative.run/)

Navigation component with toolbar

# Adding Navigation Menu in Toolbar of Application

Let's add menu components to our toolbar which provide navigation to different sections of our application.

**We'll cover the following**

* [Adding Menu components to our toolbar](https://www.educative.io/courses/getting-started-with-angular/JE27z9o2NRy#Adding-Menu-components-to-our-toolbar)
  + [Step 1: Update CustomMaterial module](https://www.educative.io/courses/getting-started-with-angular/JE27z9o2NRy#Step-1:-Update-CustomMaterial-module)
  + [Step 2: Update NavigationComponent](https://www.educative.io/courses/getting-started-with-angular/JE27z9o2NRy#Step-2:-Update-NavigationComponent)
* [Adding links to the navigation](https://www.educative.io/courses/getting-started-with-angular/JE27z9o2NRy#Adding-links-to-the-navigation)
  + [Step 1: Update NavigationComponent](https://www.educative.io/courses/getting-started-with-angular/JE27z9o2NRy#Step-1:-Update-NavigationComponent)
  + [Step 2: Update NavigationComponent class](https://www.educative.io/courses/getting-started-with-angular/JE27z9o2NRy#Step-2:-Update-NavigationComponent-class)

## Adding Menu components to our toolbar

Now, we are ready to add the menu components inside the toolbar to create our menu.

### Step 1: Update CustomMaterial module

First, we need to add MatMenuModule in the imports and exports array of CustomMaterial module (line 16 and line 24 of *custom-material.module.ts* file).

### Step 2: Update NavigationComponent

Here are the changes we’re going to make to navigation.component.html in order to add a simple menu.

📝 **Note:** Press the **RUN** button to compile and serve the application. Once the app compiles, click on the URL given after Your app can be found at, and see what happens!

###### /

e2e

src

app

navigation

navigation.component.ts

navigation.component.spec.ts

navigation.component.scss

navigation.component.html

page-not-found

custom-material.module.ts

shared

company

clients

search

app-routing.module.ts

app.component.html

app.component.scss

app.component.spec.ts

app.component.ts

app.module.ts

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.scss

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**navigation.component.html**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

<mat-toolbar>

    <p>{{ title }}</p>

    <button mat-button [matMenuTriggerFor]="clientmenu">Clients</button>

    <mat-menu #clientmenu="matMenu">

        <button mat-menu-item>Add New Client</button>

        <button mat-menu-item>View All</button>

    </mat-menu>

    <button mat-button [matMenuTriggerFor]="companymenu">Company</button>

    <mat-menu #companymenu="matMenu">

        <button mat-menu-item>Add New Company</button>

        <button mat-menu-item>View All</button>

    </mat-menu>

</mat-toolbar>





Run

Save

Reset

**Your app can be found at:**[https://ed-6434592358924288.educative.run](https://ed-6434592358924288.educative.run/)

Client Contacts Manager Application with menu items

We should see the following in our menu:

Here, we’ve added a series of menu components. <mat-menu> creates dropdown menus. In each of these, we’ve added two button components. By using the mat-menu-item directive, we are telling Angular that these buttons are part of a menu and should look and behave as drop-down menu items.

Between the two mat-menu components is another button component, but these ones are just using the mat-button directives to apply the Material style (line 4 and line 9).

When we run the application in the browser, you’ll find that we can click on one of these buttons and the menu expands, but what is making that happen?

How the menu expands on clicking the button

Well, it’s the local models we created by using the # symbol. There are two of these models, #clientmenu and #companymenu. They are used to tell Angular Material that when the button is clicked, its matMenuTriggerFor should open the corresponding menu. As you can see, the #clientmenu property is passed to the matMenuTriggerFor input of the Client’s button.

By adding these local models of #clientsmenu and #companymenu, we are creating a model that is passed into the Material Button’s matMenuTriggerFor function. When that fires off, Angular knows what model to use and display.

We now have a menu in place, alongside some buttons, but it’s currently not doing anything else.

## Adding links to the navigation

Now that the Navigation component is in place, let’s see how a user will navigate through an application. From the toolbar component, a user will click on one of the links in the component. This link will fire off a route that the user can take through the application.

### Step 1: Update NavigationComponent

Since we’ve added buttons to our menu, we are going to add click events to these buttons, which will call a function that uses the Router module to navigate to the URL we set.

The first thing we need to do is add the click events to the buttons in the navigation.component.html file:

<button mat-button [matMenuTriggerFor]="clientmenu">Clients</button>  
  <mat-menu #clientmenu="matMenu">  
    <button mat-menu-item (click)="goTo('clients')">Add New Client</button>  
    <button mat-menu-item>View All</button>  
  </mat-menu>  
  <button mat-button [matMenuTriggerFor]="companymenu">Company</button>  
  <mat-menu #companymenu="matMenu">  
    <button mat-menu-item (click)="goTo('company')">Add New Company</button>  
    <button mat-menu-item>View All</button>  
  </mat-menu>

Here, we have added two click events, and we are passing in the name of the section we want the user to go to.

### Step 2: Update NavigationComponent class

Next, we need to add this goTo() to navigation.component.ts.

📝 **Note:** Press the **RUN** button to compile and serve the application. Once the app compiles, click on the URL given after Your app can be found at, and see what happens!

###### /

e2e

src

app

navigation

navigation.component.ts

navigation.component.spec.ts

navigation.component.scss

navigation.component.html

page-not-found

custom-material.module.ts

shared

company

clients

search

app-routing.module.ts

app.component.html

app.component.scss

app.component.spec.ts

app.component.ts

app.module.ts

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.scss

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**navigation.component.ts**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

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

import { Router } from '@angular/router';

@Component({

  selector: 'app-navigation',

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

  styleUrls: ['./navigation.component.scss']

})

export class NavigationComponent implements OnInit {

  @Input()

  title: string;

  constructor(private router: Router

) {}

  ngOnInit() {}

  goTo(location: string) {

    console.log(location);

    this.router.navigateByUrl(location);

  }

}





Run

Save

Reset

**Your app can be found at:**[https://ed-6434592358924288.educative.run](https://ed-6434592358924288.educative.run/)

Client Contacts Manager Application with a navigation menu

The goTo() function works by taking in the location and using the Router class navigateByUrl() function to load the URL with the location. This location matches the path of the Routes in app-routing.module.ts. Then, Angular loads the matching path and the component of that Route.

We can also add links within pages. For these types of links, we would use routerLink. For example, imagine that we have a link from the Clients page to the Company page, but this link is with the Client view and not in the Navigation component. To do this, we would add a routerLink in the component.html file like this:

<p>  
    <a routerLink="/company">Go to the Company page</a>  
</p>

Here, we have a standard href tag, <a href=""></a>, but instead of the href attribute, we are using the routerLink directive, <a routerLink=""></a>, which has the link to the Company section. This link matches the route in the Routes module.

As you can see, there are a number of ways to trigger links within Angular. Whichever way we use the name of the route, it has to match one of our Routes in the Route module.

# Create About Section of Application

Let's add the About page to our application.

**We'll cover the following**

* [Step 1: Create AboutModule](https://www.educative.io/courses/getting-started-with-angular/B6XxJN5XE8N#Step-1:-Create-AboutModule)
* [Step 2: Create AboutPageComponent](https://www.educative.io/courses/getting-started-with-angular/B6XxJN5XE8N#Step-2:-Create-AboutPageComponent)
* [Step 3: Update AboutModule](https://www.educative.io/courses/getting-started-with-angular/B6XxJN5XE8N#Step-3:-Update-AboutModule)
* [Step 4: Update AppModule](https://www.educative.io/courses/getting-started-with-angular/B6XxJN5XE8N#Step-4:-Update-AppModule)
* [Step 5: Create a new route](https://www.educative.io/courses/getting-started-with-angular/B6XxJN5XE8N#Step-5:-Create-a-new-route)
* [Step 6: Update NavigationComponent](https://www.educative.io/courses/getting-started-with-angular/B6XxJN5XE8N#Step-6:-Update-NavigationComponent)
* [Step 7: Update AboutComponent](https://www.educative.io/courses/getting-started-with-angular/B6XxJN5XE8N#Step-7:-Update-AboutComponent)
* [About section in action](https://www.educative.io/courses/getting-started-with-angular/B6XxJN5XE8N#About-section-in-action)

Let’s discuss the steps you’ll need to take to add this About section to our application

## Step 1: Create AboutModule

First, you have to navigate to the app folder and create a new about module using the following command:

ng generate module about

📝 **Note:** Click the terminal window and see what happens!

This code requires the following environment variables to execute:

NG\_CLI\_ANALYTICS

off

npm\_config\_loglevel

silent

CHROME\_BIN

usr/bin/google-chrome

Edit

**Terminal 1**

Terminal

**Click to Connect...**

****

## Step 2: Create AboutPageComponent

Navigate to the about folder and create a new AboutPage component by running the following command in the terminal:

ng generate component about-page --skip-import

## Step 3: Update AboutModule

Add the new AboutPage component to the AboutModule:

declarations: [AboutPageComponent],

Also, add CustomMaterialModule in the AboutModule:

imports: [  
    CommonModule,  
    CustomMaterialModule  
  ]

## Step 4: Update AppModule

Now open app.module.ts and add AboutModule to the imports array:

imports: [  
    BrowserModule,  
    AppRoutingModule,  
    BrowserAnimationsModule,  
    ClientModule,  
    CompanyModule,  
    SharedModule,  
    CustomMaterialModule,  
    AboutModule,   
  ],

## Step 5: Create a new route

Add a new route to the app-routing.module.ts file:

{  
  path: 'about',  
  component: AboutPageComponent  
},

## Step 6: Update NavigationComponent

Now, add the About button in a navigation.component.html file:

<button mat-button (click)="goTo('about')">About</button>

## Step 7: Update AboutComponent

Add an Angular Material component to display some text about this application in about.component.html file:

<h2>About</h2>  
<mat-card>  
  <mat-card-content>  
    This is the Client Contacts Manager, where you can add the contact details  
    of your sales clients.  
  </mat-card-content>  
</mat-card>

## About section in action

📝 **Note:** Press the **RUN** button to compile and serve the application. Once the app compiles, click on the URL given after Your app can be found at, and see what happens!

###### /

e2e

src

app

about

about-page

about-page.component.ts

about-page.component.spec.ts

about-page.component.scss

about-page.component.html

about.module.ts

navigation

page-not-found

custom-material.module.ts

shared

company

clients

search

app-routing.module.ts

app.component.html

app.component.scss

app.component.spec.ts

app.component.ts

app.module.ts

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.scss

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**about-page.component.html**

1

2

3

4

5

6

7

8

<h2>About</h2>

<mat-card>

  <mat-card-content>

    This is the Client Contacts Manager, where you can add the contact details

    of your sales clients.

  </mat-card-content>

</mat-card>





Run

Save

Reset

**Your app can be found at:**[https://ed-6434592358924288.educative.run](https://ed-6434592358924288.educative.run/)

Client Contacts Manager Application with about section

You should see the following in the About section:

Client Contacts Manager Application with About section

# Route Parameters

Let's explore how we can use route parameters to pass an ID or parameter as part of the URL and load data based on that parameter.

**We'll cover the following**

* [What are Route parameters?](https://www.educative.io/courses/getting-started-with-angular/q2GwLZpg8zD#What-are-Route-parameters?)
  + [Example](https://www.educative.io/courses/getting-started-with-angular/q2GwLZpg8zD#Example)
* [Setting up a route to handle parameters](https://www.educative.io/courses/getting-started-with-angular/q2GwLZpg8zD#Setting-up-a-route-to-handle-parameters)
* [Accessing Route parameters](https://www.educative.io/courses/getting-started-with-angular/q2GwLZpg8zD#Accessing-Route-parameters)
  + [Create a private argument for the ActivatedRoute class](https://www.educative.io/courses/getting-started-with-angular/q2GwLZpg8zD#Create-a-private-argument-for--the-ActivatedRoute-class)
  + [Use snapshot property](https://www.educative.io/courses/getting-started-with-angular/q2GwLZpg8zD#Use-snapshot-property)
  + [Use paramMap function](https://www.educative.io/courses/getting-started-with-angular/q2GwLZpg8zD#Use-paramMap-function)
  + [Display route parameter](https://www.educative.io/courses/getting-started-with-angular/q2GwLZpg8zD#Display-route-parameter)
* [Route parameter in action](https://www.educative.io/courses/getting-started-with-angular/q2GwLZpg8zD#Route-parameter-in-action)

A very common part of any modern web application is passing an ID or parameter as part of the URL to load data based on that parameter. In Angular, this is achieved using Route parameters.

## What are Route parameters?

**Route parameters** are dynamic values that are attached to the end of a URL. This value is then used to load data or a flag in the view to change how a view will look.

You would probably use this to load in the details of a specific client or company, something we will be doing once we have built up the application enough to start saving data.

### Example

For example, here is a standard URL: http://localhost:4200/clients. A URL with a parameter on the end of it would look like this: http://localhost:4200/client/24. The difference is that at the end of the second URL is a number, which could be the ID of a Client whose details we want to load. This is something we will be doing in the next chapter.

URL with parameter on the end

## Setting up a route to handle parameters

In order to access this ID, we need to add a route that is able to handle the fact that the ID is part of the URL. To do this, we have to add another route to the Routes array in the app-routing.module.ts file.

Now, we need to add this new route that can handle the ID part of the URL.

{  
    path: 'clients/:id',  
    component: ClientPageComponent  
},

The only change we’ve made is adding this new Clients route, with /:id at the end. Now, when the Route sees the URL with the ID at the end, it knows it has a route that matches the signature of the URL. If we didn’t do this, Angular wouldn’t be able to match the URL with one of the Routes, and it would go to the empty path route, which in this case would take the user to the /clients page.

By setting up this extra route, the Angular Router knows what to do if it sees a URL with the ID. However, this is only half of the story. Now, we need to be able to access the ID.

## Accessing Route parameters

There are a few approaches you can take to access the Route parameters. The approach we are going to use allows you to access the ID from the URL. There are other ways we can access the parameters from the URL, but these involve using RxJS, which we will explore in the [Observables and RxJs chapter](https://www.educative.io/collection/page/10370001/4603693004488704/4959879964917760).

We’re going to access the ID in the ClientPage component so that it is available at the parent level and can be passed into the Child components if they need access to this data. First, we need to open the client-page.component.ts file. Here, we can use the ngOnInit life cycle handler to access the route parameters at the time the ClientPage component is being initialized/created.

Let’s update the ClientPageComponent class so that it looks like this:

export class ClientPageComponent implements OnInit {  
  constructor(private route: ActivatedRoute) {}  
  
  ngOnInit() {  
    let id = this.route.snapshot.paramMap.get('id');  
    console.log('Our passed ID is', id);  
  }  
}

### Create a private argument for the ActivatedRoute class

There’s a couple of things going on here. First, we’ve created a new private argument to the constructor for the ActivatedRoute class, which is a class from the Router module of Angular. This gives us a set of functions we can use when working with the URL.

📝 **Note:** To find out more about what the ActivatedRoute class provides, read the [official Angular documentation](https://angular.io/api/router/ActivatedRoute).

### Use snapshot property

The next change we’ve made is updating the ngOnInit() function to create a local variable called id, then using the private route property we created in the constructor to use the snapshot property of the ActivatedRoute class. This snapshot property is an interface itself and contains information about the route. It gives us a snapshot of the route at the time it’s loaded. From this, we can access the ID that is passed in the URL.

### Use paramMap function

The ActiviatedSnapshotRoute interface has information about a route associated with a component loaded in an outlet currently.

We can use the paramMap property of the ActiviatedSnapshotRoute interface that provides a map to access all the parameters in the URL (a URL could have more than one parameter to pass).

This mapping allows us to get each individual parameter using the get() method by passing in the name of the parameter we want. So, if the URL had both ID and name as two separate parameters, we could use paramMap to get('id') and get('name') as two more separate parameters.

Accessing route parameters

### Display route parameter

Finally, we would use the console.log statement to display the Router parameter in the browser console.

## Route parameter in action

We can test that this works by changing the ID at the end of the URL.

📝 **Note:** Press the **RUN** button to compile and serve the application. Once the app compiles, click on the URL given after Your app can be found at, and change the parameter from 24 to 25 and observe the output of the console!

###### /

e2e

src

app

about

navigation

page-not-found

custom-material.module.ts

shared

company

clients

client.module.ts

client-form

client-page

client-page.component.html

client-page.component.scss

client-page.component.spec.ts

client-page.component.ts

search

app-routing.module.ts

app.component.html

app.component.scss

app.component.spec.ts

app.component.ts

app.module.ts

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.scss

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**client-page.component.ts**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

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

import { Router, ActivatedRoute, ParamMap } from '@angular/router';

@Component({

  selector: 'app-client-page',

  templateUrl: './client-page.component.html',

  styleUrls: ['./client-page.component.scss']

})

export class ClientPageComponent implements OnInit {

  constructor(private route: ActivatedRoute) {}

  ngOnInit() {

    let id = this.route.snapshot.paramMap.get('id');

    console.log('Our passed ID is', id);

  }

}





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/clients/24>

Client Contacts Manager Application with route parameter

You will see the following in the console:

Console output after changing the client id

Now, we can access parameters that are passed via a URL. This will allow our application to use this data to filter lists by ID, which is a common pattern in modern web applications.

# Route Guards

Familiarize yourself with route guards and explore the scenarios in which we can use them.

**We'll cover the following**

* [What is the route guard?](https://www.educative.io/courses/getting-started-with-angular/Bn6y441YjWW#What-is-the-route-guard?)
* [Scenarios for using a route guard](https://www.educative.io/courses/getting-started-with-angular/Bn6y441YjWW#Scenarios-for-using-a-route-guard)
* [Guard interface](https://www.educative.io/courses/getting-started-with-angular/Bn6y441YjWW#Guard-interface)

## What is the route guard?

Currently, all our routes are accessible by anyone using the application. For example, a user could go to the Clients section or the Company section and see all the data that’s there, but this is something we might not want to happen in a real-world application. Say, for example, I have a set of Clients assigned to me, and you, as another member of the sales team, have a different set. I wouldn’t want you to be able to go to the application, load up the Clients section, and then see all of my Clients.

We can prevent this through the use of Route Guards. Route Guards are classes that implement a set of five different interfaces.

**Angular’s route guards** are interfaces that can tell the router whether or not it should allow navigation to a requested route.

Route guard

📝 **Note:** An interface is a class that has no implementation, just a set of empty functions, but if we create a class that implements the interface, our class needs to have functions that match the signature of all the functions defined in the interface. If not, then our class has not fully implemented the interface.

## Scenarios for using a route guard

There are a number of scenarios when you would use a route guard. For example:

* Checking that the user has logged into the application before going to a route
* Checking that the user has the right permissions to go to a certain route
* Loading data before a route has fully loaded
* Saving data when going from one route to another
* Checking whether the user wants to save any unsaved data before leaving to go to another route

## Guard interface

As you can see, route guards can be very helpful. In order to cover these different use cases, Angular has a set of five guard interfaces. These are as follows:

| **Guard interface** | Purpose |
| --- | --- |
| **CanActivate** | Check that access is allowed before going to a route |
| **CanActivateChild** | Check that access is allowed before going to a child route |
| **CanDeactivate** | Decides if a route can be deactivated |
| **Resolve** | Retrieves data before loading a route |
| **CanLoad** | Checks that access is allowed before loading a module |

While each Guard handles different use cases, we can have a class that implements multiple route guards, so we could create a guard that checks to see if we have access to a route. If so, it will load data before loading the route.

# Implementing a Route Guard

Let's explore how we can use a route guard to check if the user is logged in before going to the route.

**We'll cover the following**

* [Step 1: Create a TypeScript class](https://www.educative.io/courses/getting-started-with-angular/gxLPOxRpDo9#Step-1:-Create-a-TypeScript-class)
* [Step 2: Add guard to the route](https://www.educative.io/courses/getting-started-with-angular/gxLPOxRpDo9#Step-2:-Add-guard-to-the-route)
* [Step 3: Add Guard to AppModule](https://www.educative.io/courses/getting-started-with-angular/gxLPOxRpDo9#Step-3:-Add-Guard-to-AppModule)

## Step 1: Create a TypeScript class

To implement a route guard, we need to create a TypeScript class that uses one or more of these route interfaces. By implementing an interface, we need to have a function in our class that the interface says we should have.

So, if we implement the CanActivate interface, we need to have a canActivate() function in our class. This function returns true or false. If true, the user can access the route; if false, then they can’t.

canActivate( ) function working

For example, here is a class that uses a service (we will be looking at this in the [Dependency Injection, Services, and HttpClient chapter](https://www.educative.io/collection/page/10370001/4603693004488704/6233226233249792)) to authenticate the user to see if they can activate a route:

import { Injectable } from '@angular/core';  
import { CanActivate } from '@angular/router';  
import { AuthService } from './auth.service';  
  
@Injectable()  
export class CanActivateViaAuthGuard implements CanActivate {  
     constructor(private authService: AuthService) {}  
     canActivate() {  
         return this.authService.isLoggedIn();  
    }  
}

Here, we are creating a new class called CanActivateViaAuthGuard, which implements the CanActivate router guard interface. By doing so, the class needs to have the canActive() function.

In the canActivate() function, we are making a call to an authentication service to check if the user is logged in. This isLoggedIn() function would need to return either true or false.

## Step 2: Add guard to the route

Once we have this guard, we need to add it to the route we want to check. We do this by adding the CanActivateViaAuthGuard class to the route in the Routes array of our app-routing.module.ts file. So, if we were going to add this CanActivateViaAuthGuard to the Clients route, it would look like this:

{  
    path: 'client',  
    component: ClientPageComponent,  
    canActivate: [  
        'CanAlwaysActivateGuard',  
        CanActivateViaAuthGuard  
    ]  
}

Here, we are stating that this Route is using the canActivate guard by providing the name and title of the Guard to be used by the Route when it runs its canActivate() function.

## Step 3: Add Guard to AppModule

The final step of adding in a route guard is to add the Guard class to the list of providers in our AppModule so that Angular knows where to find the route guard class:

@NgModule({  
     ...  
    providers: [  
        AuthService,  
        CanActivateViaAuthGuard  
    ]   
})  
export class AppModule {}

In the providers array, we are setting both the guard and the service we are using to check that a user has access.

As you can see, implementing a Route Guard is quite straightforward. You create a new TypeScript class, implement one or more Route Guard interfaces, and add the implementation of the function the interface requires. Then, you register the Guard with the AppModule providers array and add the Guard class as the Guard for the route.

# Dependency Injection, Services, and HttpClient

Let's learn about Dependency Injection, Services, and HttpClient in this chapter.

**We'll cover the following**

* [Chapter learning outcomes](https://www.educative.io/courses/getting-started-with-angular/m7QxKMYwq1A#Chapter-learning-outcomes)
* [Client contacts manager application](https://www.educative.io/courses/getting-started-with-angular/m7QxKMYwq1A#Client-contacts-manager-application)

## Chapter learning outcomes

In this chapter, we are going to start looking at some more advanced topics of Angular development. So far, we have looked at how an Angular application is structured, how Angular components are created, and how we use the Angular CLI for almost everything we need to do in order to build an application. We’ve explored how to add a UI to an application and how we can add Angular Material to give a polished look and feel to the application. Now, we need to start looking at **how we can add data to an Angular application**.

Over the next few chapters, we will be looking at **how Angular manages data**, how it loads external data, and how it manages the concept of the state within an application.

In this chapter, we will start by looking at **services**, what they are, and how to make use of them in our Angular applications.

By the end of this chapter, you will have learned the following topics:

* What is Dependency Injection?
* Why we use Dependency Injection in Angular.
* What services and providers are in Angular.
* How services are used in Angular and how components make use of services.
* How to load data from an external source using the built-in services that Angular provides.

## Client contacts manager application

We will also be expanding on our demo application to make use of services in order to save and manage data within the application.

📝 **Note:** Press the **RUN** button to compile and serve the application. Once the app compiles, you can see its output in the **Output** tab or by clicking on the URL given after Your app can be found at.

###### /

e2e

src

app

about

clients

client-detail

client-details-page

client-edit-page

client-form

client-list

client-page

client-search-page

client.module.ts

client.service.spec.ts

client.service.ts

client.ts

company

navigation

page-not-found

search

services

shared

app-routing.module.ts

app.component.html

app.component.scss

app.component.spec.ts

app.component.ts

app.module.ts

custom-material.module.ts

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.scss

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**client.module.ts**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

import { CommonModule } from '@angular/common';

import { CUSTOM\_ELEMENTS\_SCHEMA, NgModule } from '@angular/core';

import { FormsModule, ReactiveFormsModule } from '@angular/forms';

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

import { CustomMaterialModule } from '../custom-material.module';

import { SharedModule } from '../shared/shared.module';

import { ClientDetailComponent } from './client-detail/client-detail.component';

import { ClientDetailsPageComponent } from './client-details-page/client-details-page.component';

import { ClientEditPageComponent } from './client-edit-page/client-edit-page.component';

import { ClientFormComponent } from './client-form/client-form.component';

import { ClientItemComponent } from './client-list/client-item/client-item.component';

import { ClientListComponent } from './client-list/client-list.component';

import { ClientPageComponent } from './client-page/client-page.component';

import { ClientSearchPageComponent } from './client-search-page/client-search-page.component';

import { ClientService } from './client.service';

const routes: Routes = [

  {

    path: 'clients/new',

    component: ClientPageComponent

  },

  {

    path: 'clients/search',

    component: ClientSearchPageComponent

  },

  {

    path: 'clients/details/:id',

    component: ClientDetailsPageComponent

  },

  {

    path: 'clients/edit/:id',

    component: ClientEditPageComponent





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/>

Client Contacts Manager Application with Services

In this version of application, we have made the following changes:

* Added services, one for the Client section and one for the Company section.
* Added forms for adding, editing, and deleting both Clients and Companies.
* Used [angular-in-memory-web-api](https://github.com/angular/in-memory-web-api) to store data locally.
* Added lazy loading of routes.

# What Is Dependency Injection?

Let's explore what dependency injection is and what benefits it brings to our application.

**We'll cover the following**

* [What is Dependency Injection?](https://www.educative.io/courses/getting-started-with-angular/39nr08zr0MA#What-is-Dependency-Injection?)
  + [Example](https://www.educative.io/courses/getting-started-with-angular/39nr08zr0MA#Example)
* [The benefits of Dependency Injection](https://www.educative.io/courses/getting-started-with-angular/39nr08zr0MA#The-benefits-of-Dependency-Injection)
  + [Allows us to divide a task into many different classes](https://www.educative.io/courses/getting-started-with-angular/39nr08zr0MA#Allows-us-to-divide-a-task-into-many-different-classes)
  + [Makes it easier to swap our classes](https://www.educative.io/courses/getting-started-with-angular/39nr08zr0MA#Makes-it-easier-to-swap-our-classes)
  + [Separation of concerns](https://www.educative.io/courses/getting-started-with-angular/39nr08zr0MA#Separation-of-concerns)

Before we start making changes to our application, we need to go through some new topics that we have not discussed yet, starting with Dependency Injection.

## What is Dependency Injection?

**Dependency Injection** is a design pattern that sets out how objects are passed to one another. **Design patterns are reusable solutions** to common problems we find in developing software. There are many different types of design patterns, which the Dependency Injection is part of. If you spend time looking at these different patterns, they can provide possible solutions to issues you may come across in your development career. So, they are worth exploring.

The Design Pattern of **Dependency Injection** states that a class can ask for its dependencies to be passed in; instead of creating instances of its dependencies itself.

### Example

For example, we have a class (ClassA) that needs to use two other classes, one to load some data (ClassB) and the other (ClassC) to sort some data. Instead of ClassA having to create an instance of ClassB and one of ClassC before it uses them, ClassA can use some of ClassB functionality without the overhead of having to create instances of these other classes.

Dependency injection

This is a high-level overview of what Dependency Injection is. This ability of a class to have the other classes it is dependent on injected into itself instead of having to manually create the dependencies has benefits that make it ideal for Angular.

## The benefits of Dependency Injection

Let’s go through the advantages of dependency injection.

### Allows us to divide a task into many different classes

The benefit of **Dependency Injection (DI)** is that it allows us to divide a task into many different classes that, through DI, are easy for the class using these services to ingest.

Again, going back to our ClassA example, if this class needs to handle a piece of complex functionality – user login, for example – ClassA might need to access an external API, check the results of an API call, store a record in local storage, update the UI, or redirect the user if login fails.

There could be a number of functions that this ClassA needs to do. In order to make our code cleaner, we should **follow the principle of single-responsibility**, where a class does one thing and one thing only. We need to share this functionality across a number of classes, each doing one thing (for instance, storing data in local storage). This should lead to a number of smaller classes, each doing one thing well.

Single responsibility principle

But if ClassA needs all these other classes, it would have a lot of overhead creating these classes. With DI, ClassA just calls in the classes it needs. There is no need to create instances of the class. So, from an architectural point of view, we can create as many classes as we need, each class doing only one thing, **making them easier to test and easier to understand what this class does**.

### Makes it easier to swap our classes

Another great benefit of DI is that it makes it easier to swap out classes, which is very helpful when it comes to testing. If in ClassA, we inject a ClassC that returns data from an external API. In our unit tests, we can inject in a mock ClassC that returns mock data, removing the need to load data from an external API in order to test the functionality of ClassA. This approach means **we can take individual elements/objects and test them in isolation**. We will go through this in more depth when we go through testing in the [Testing chapter](https://www.educative.io/collection/page/10370001/4603693004488704/5661344538820608).

Test individual elements in isolation

### Separation of concerns

Other great benefits of DI include this separation of concern, where **each part of an application is not aware of other parts of the application**. So, if we need to make a change or refactor a piece of the application, it’s being separated from the rest means that making a change in one place has less chance of breaking something somewhere else (we say less chance because there is always a slight chance a change will have an uninteded effect, but this is why we have Unit Tests).

Separation of concerns

# How Does Angular Handle Dependency Injection?

Familiarize yourself with the Dependency Injection framework and create a client service for the Client Contacts Manager application using the Angular CLI.

**We'll cover the following**

* [Dependency Injection framework](https://www.educative.io/courses/getting-started-with-angular/q2KEK85RBJr#Dependency-Injection-framework)
* [Creating an Injectable service](https://www.educative.io/courses/getting-started-with-angular/q2KEK85RBJr#Creating-an-Injectable-service)
* [Setup of service with an Injector](https://www.educative.io/courses/getting-started-with-angular/q2KEK85RBJr#Setup-of-service-with-an-Injector)

## Dependency Injection framework

The **Dependency Injection framework**, which handles how classes are injected when needed, is part of the larger Angular framework.

As Angular developers, we don’t need to worry about how Angular handles DI. All we need to be concerned about is how we structure our application to make use of DI.

When building our Angular applications, we need to remember that through the use of DI, we can and should **use a more modularized approach to how we structure our application**, making use of services and creating common classes that can be used many times throughout our application, and providing components with everything they need. All of this makes our applications as efficient as possible.

Create more modularity and reusability within the application

Angular provides the Dependency Injection framework for us. All we need to do is to set up our classes to use the DI framework, which is what we will look at now.

## Creating an Injectable service

Like so many things in Angular, we turn to the Angular CLI to create a service for us. For example, to create a service in our Client section of the Client Contact Manager Application, in the terminal we would navigate to the client folder then run this command:

ng generate service client

This will tell the Angular CLI to create a service class for us, using the client argument as the title of the service. This is the file it has created for us:

###### /

e2e

src

app

about

navigation

page-not-found

custom-material.module.ts

shared

company

clients

client.service.ts

client.service.spec.ts

client.module.ts

client-form

client-page

search

app-routing.module.ts

app.component.html

app.component.scss

app.component.spec.ts

app.component.ts

app.module.ts

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.scss

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**client.service.ts**

1

2

3

4

5

6

7

8

9

10

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

@Injectable({

  providedIn: 'root'

})

export class ClientService {

  constructor() { }

}





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/>

Client Contacts Manager Application with ClientService

There’s not much here, but this is the beginning of a service. The next stage in the Dependency Injection process is to set this class as an injectable. To do that, we use an Injector.

## Setup of service with an Injector

In the preceding ClientService example, you can see we are using the @Injectable operator at the top of our class. As we have seen with other decorators such as @NgModule and @Component, this is used to tell Angular something about this class. In this instance, the @Injectable decorator tells Angular that the class is designed to be injected into another class. That is all this operator is doing: making Angular aware that we want our ClientService class to be injected.

# The Hierarchy of Injectors

Explore how Angular employs a hierarchical system to use the different types of injectors.

**We'll cover the following**

* [Injectors in Angular](https://www.educative.io/courses/getting-started-with-angular/x1Q5ZKmx2WB#Injectors-in-Angular)
* [Injectors hierarchy](https://www.educative.io/courses/getting-started-with-angular/x1Q5ZKmx2WB#Injectors-hierarchy)
* [Example](https://www.educative.io/courses/getting-started-with-angular/x1Q5ZKmx2WB#Example)
  + [The @Component level Injector](https://www.educative.io/courses/getting-started-with-angular/x1Q5ZKmx2WB#The-@Component-level-Injector)
  + [The @Injectable level injector](https://www.educative.io/courses/getting-started-with-angular/x1Q5ZKmx2WB#The-@Injectable-level-injector)
  + [The @NgModule level injector](https://www.educative.io/courses/getting-started-with-angular/x1Q5ZKmx2WB#The-@NgModule-level-injector)

## Injectors in Angular

We have seen three types of decorators:

Injectors in Angular

These injectors are used to tell Angular something about the class.

## Injectors hierarchy

With these three different injectors knowing which ones to use and when can be confusing at first. Thankfully Angular employs a hierarchical system for how it uses the different injectors.

When the Dependency Injection framework in Angular needs to find a class that is being injected into a component, it starts by looking at the @Component operator-level injector. If the requested class cannot be found there, Angular moves on to the @Injectable operator level and trys to find the class being referred to. If Angular still cannot find the class being searched for at this level, Angular moves on to looking at the@NgModule operator level to see if the class is referenced there.

## Example

Let’s have a look at an example of how this works. If we have a component that needs a service to be injected, we use the following code:

###### /

e2e

src

app

about

navigation

page-not-found

custom-material.module.ts

shared

company

clients

client.ts

client.service.ts

client.service.spec.ts

client.module.ts

client-form

client-page

client-page.component.html

client-page.component.scss

client-page.component.spec.ts

client-page.component.ts

search

app-routing.module.ts

app.component.html

app.component.scss

app.component.spec.ts

app.component.ts

app.module.ts

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.scss

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**client-page.component.ts**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

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

import { Client } from '../client';

import { ClientService } from '../client.service';

@Component({

  selector: 'app-client-page',

  templateUrl: './client-page.component.html',

  styleUrls: ['./client-page.component.scss'],

  providers: [ClientService]

})

export class ClientPageComponent implements OnInit {

  constructor(private clientService: ClientService) {}

  ngOnInit() { }

  saveClient(clientDetails: Client) {

      console.log('clientDetails', clientDetails);

      this.clientService.save(clientDetails);

  }

}





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/>

Client Contacts Manager Application with ClientService injected into the ClientPage component

In this ClientPageComponent, we are injecting **ClientService** in the constructor. Then we can use the save() function of the ClientService in the saveClient() method of ClientPageComponent. For this to work, Angular needs to find this ClientService. So, through the hierarchical approach for Injectors, Angular will first look to see if ClientService is set at the @Component operator level (which in this example it’s not). If it is not, Angular will then move on to the @Injectable operator level, looking through all the @Injectable operators it is aware of.

If we look at the ClientService code, we can see that the @Injectable operator is there, so Angular knows that this class is injectable:

@Injectable({  
       providedIn: 'root'  
})  
export class ClientService {  
    constructor() { }  
  
    // save a client  
    save(client: Client) {  
        // ...   
    }  
}

But if this class didn’t have the @Injectable operator, Angular would move onto the next level up, which is at the @NgModule level. So, you see, Angular uses these levels to search up the codebase to find the classes we are referencing. Let’s have a closer look at each of these levels to see how a class is set to be injected at each level.

How a Dependency Injection framework finds a class

### The @Component level Injector

We’ve already seen the structure of the @Component operator in [Components, Templates, and Forms chapter](https://www.educative.io/collection/page/10370001/4603693004488704/5860980926971904), and how we set the template of a component and the styling files for it. But there is another part of the @Component operator that allows users to set a list of classes that the component needs to be provided with.

This is the **provider’s array**. If we take a look at our ClientPageComponent example with the providers array:

@Component({  
    selector: 'app-client-page',  
    templateUrl: './client-page.component.html',   
    styleUrls: ['./client-page.component.scss'],   
    providers: [ClientService]  
   })  
   export class ClientPageComponent implements OnInit {  
   ...

Now, in the example, we have set it so that our ClientService is providing to the component through its own providers array.

###### Question

What if the other two levels of injector would want to use this level?

Show Answer

### The @Injectable level injector

When we created our ClientService service using the Angular CLI, it added the @Injectale operator for us at the top of the class. As part of this, it provided an object with a provideIn property set:

@Injectable({  
       providedIn: 'root'  
})

This provideIn property, which is set to root, is telling Angular to use the root module to automatically provide this class. But we may decide that the class we create should only be a certain module, for example, CompanyModule. So, we could change provideIn to this:

@Injectable({  
    providedIn: CompanyModule  
})  
export class ClientService {  
   ...

Now the ClientService is only available to any components that are part of CompanyModule (in a real application, it would be the ClientModule), so if we try to add the ClientService into a component outside of the CompanyModule, the Angular compiler would throw an error and warn us that ClientService is not available.

### The @NgModule level injector

The final level of the injector is at the @NgModule level. We’ve already been through the structure of the @NgModule class in the [NgModule chapter](https://www.educative.io/collection/page/10370001/4603693004488704/4518520179130368" \t "_blank). In that chapter, we went through all the parts of the module class: one of these parts was the providers array. This, like the provider array at the @Component level, allows us to set what classes are available to be injected anywhere across the entire module.

The following code shows us how that works in our ClientModule:

@NgModule({  
       declarations: [  
           ClientPageComponent,   
           ClientSearchPageComponent,  
           ClientFormComponent,   
           ClientListComponent  
       ],  
       imports: [  
           CommonModule,   
           ReactiveFormsModule,   
           SharedModule,  
           CustomMaterialModule  
       ],  
       providers: [ClientService]  
   })  
   export class ClientModule {}

You can see here that ClientService is now provided at the module level. So, ClientService is available to be injected into all components, directives, and other classes that are part of this module.

# Benefits of Three Levels of Injectors

Let's discuss the benefits of having these three levels of the injectors as Angular developers.

**We'll cover the following**

* [Isolation of concerns](https://www.educative.io/courses/getting-started-with-angular/39jV7vwJRY9#Isolation-of-concerns)
  + [Example](https://www.educative.io/courses/getting-started-with-angular/39jV7vwJRY9#Example)
* [Tree shaking](https://www.educative.io/courses/getting-started-with-angular/39jV7vwJRY9#Tree-shaking)
  + [Using providedIn property for services](https://www.educative.io/courses/getting-started-with-angular/39jV7vwJRY9#Using-providedIn-property-for-services)

## Isolation of concerns

Well, having the ability to set that a class can be injected at different levels allows us to **create certain levels of isolation**. We can, as we briefly discussed in the @Component Injector section, set it so that a service can only be injected at a component level or is only available at a certain module level.

This means we can **create specialized services that perform one task, and are only needed in one place**, if our Angular application needs to provide some business logic that is critical to the application but is very specialized.

Isolation of concerns

### Example

For example, if our application had to do with finance, we may have a service for working out client financial statements. This is a specialized service that is only needed in one part of the application. Therefore we need to limit the exposure of service to other parts of the application so that it’s clear that this service only does one job in the one place of our application. This isolation of concerns is a good practice to use in applications.

## Tree shaking

One of the main benefits of using Injectors is the ability for the Angular compiler to use **tree shaking** when running the application. The idea behind tree shaking is that **when an application is running, any services that aren’t used at the time can be removed from the final bundle of the application**. The Angular compiler shakes the application to remove all the dead leaves (or services in Angular’s case). That’s why this feature is called tree shaking.

This has the benefit of creating a download that is as small as possible when the user runs the appication. The less the user has to download before the application starts, the **faster the application will startup**.

Tree shaking in Angular

### Using providedIn property for services

What helps Angular know about these services is the providedIn property we set in the @Injector operator of our ClientService.

@Injectable({  
  providedIn: 'root'  
})

In previous versions of Angular, where tree shaking was not a feature, there was a one-way connection between the services being injected and the providers. The provider contained a list of the services within the module, and the service did not know what module it belonged to.

Now, with the providedIn property, **the service also knows what module it belongs to**. So, the module no longer needs to know of all the dependencies, and therefore when the application is running, the unused dependencies (services and so on) can be removed. This new way of setting dependencies makes tree shaking possible in the latest version of Angular.

We will be covering bundling and deploying an application later in the [Packaging our application chapter](https://www.educative.io/collection/page/10370001/4603693004488704/6173970884722688).

# Providers in Angular

Let's go through all the different approaches for creating providers in Angular.

**We'll cover the following**

* [What is the provider?](https://www.educative.io/courses/getting-started-with-angular/q2r1VkGqqr3#What-is-the-provider?)
* [Provider object literal](https://www.educative.io/courses/getting-started-with-angular/q2r1VkGqqr3#Provider-object-literal)
* [Alternative class providers](https://www.educative.io/courses/getting-started-with-angular/q2r1VkGqqr3#Alternative-class-providers)
* [Value providers](https://www.educative.io/courses/getting-started-with-angular/q2r1VkGqqr3#Value-providers)
* [Factory providers](https://www.educative.io/courses/getting-started-with-angular/q2r1VkGqqr3#Factory-providers)
* [Predefined tokens and providers](https://www.educative.io/courses/getting-started-with-angular/q2r1VkGqqr3#Predefined-tokens-and-providers)

## What is the provider?

So, we’ve had a look at what Injectors are and the various levels of Injectors in Angular, but now we are going to have a look at providers. We have seen them briefly already, but it’s worth having a more in-depth look at them and the role they fulfill in Angular.

The Injector needs a **provider** to create the instances (versions) of the classes being injected into all the components, services, and directives.

The way this relationship between injector and provider is set up is through a token that the injector uses at runtime to create the required class that the provider needs. So, when our application is running and a class is needed via Dependency Injection, the token that the provider gives makes sure that the class being used is the one with the correct token.

Relationship between provider and injector

In ClientModule, we can see how ClientService is itself being used as this token:

@NgModule({  
  declarations: [ClientPageComponent, ClientFormComponent],  
  providers: [ClientService],  
  imports: [CommonModule,   
            ReactiveFormsModule,   
            SharedModule,   
            CustomMaterialModule],  
})  
export class ClientModule {}

Here, ClientService is the token that is being provided to the providers array.

## Provider object literal

The provider object literal will allow us to give a more relevant name to our services.

We could also write the name of the service in another way, by using an object to set ClientService as a token for the provider **(line 12)**:

###### /

e2e

src

app

about

navigation

page-not-found

custom-material.module.ts

shared

company

clients

client.ts

client.service.ts

client.service.spec.ts

client.module.ts

client-form

client-page

search

app-routing.module.ts

app.component.html

app.component.scss

app.component.spec.ts

app.component.ts

app.module.ts

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.scss

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**client.module.ts**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

import { NgModule } from "@angular/core";

import { CommonModule } from "@angular/common";

import { ClientPageComponent } from "./client-page/client-page.component";

import { ClientFormComponent } from "./client-form/client-form.component";

import { ReactiveFormsModule } from "@angular/forms";

import { SharedModule } from '../shared/shared.module';

import { CustomMaterialModule } from '../custom-material.module';

import { ClientService } from './client.service';

@NgModule({

  declarations: [ClientPageComponent, ClientFormComponent],

  providers: [{provide: ClientService, useClass: ClientService}],

  // providers: [ClientService],

  imports: [CommonModule, ReactiveFormsModule, SharedModule, CustomMaterialModule],

})

export class ClientModule {}





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/>

Client Contacts Manager Application with ClientService being injected into the module

Why would you use this option? Well, you may have a service that you haven’t written but you need to give it a more relevant name. For example, a third-party service called HttpClient isn’t a great name, but we could use this approach to give it a more relevant name to our application:

[{provide: HttpClientService, useClass: HttpClient}]

It allows us to use HttpClientService, in our components, instead of a name like HttpClient, which doesn’t give a clear idea of what the component does. This approach is called the **provider object literal** and it helps if we are using external libraries, which we are.

## Alternative class providers

Alternative class providers will allow you to use a different class but for the same provider token.

Now, sometimes we may want to use a different class for a provider but still use the name of a provider that has already been established. For example, you are working on a large application and there is already a service that is part of your codebase called EmailScheduler, but in the section of the application you are working on, you need a more specific email scheduling class. So, you decide to create one that extends upon the original EmailScheduler.

Therefore, you create a new class called CompanyEmailScheduler and then need to add it to your module. The problem is that there is a component you are using that is already using the original EmailScheduler. So what do you do to start renaming this EmailScheduler class everywhere that it is referenced?

Well, in this instance you can use an **Alternative Class Provider**. This will allow you to use a different class but for the same provider token. The syntax for this alternative class provider is the same as the previous example:

[{provide: EmailScheduler, useClass: CompanyEmailScheduler}]

This has allowed us to create a new class without breaking the existing codebase. Our module will work using our new CompanyEmailScheduler class, which extends on the original class. The component that has the original EmailScheduler as a dependency is still going to work.

## Value providers

There is another way to create a provider: through value providers.

**Value providers** give us a way to supply an object instead of an Injector to the Provider.

We may use this approach if we want to provide a simple exported function that performs a single task.

In this example, we’re creating a simple function, then setting it to a variable that is used in an object literal when we set the provider:

// exported function that performs an action  
export function SimpleFn() {  
   console.log('This is a simple function');  
}  
  
// object literal  
const simpleFunctionProvider {  
    simpleFn: SimpleFn  
};  
  
// setting the provider  
[{provider: SimpleFunc, useClass: SimpleFn }]

The Value Provider allows users to create a small function that can be used as a Provider. If we wanted to have a Provider that provides a piece of information as a string, we could use the value Provider to do this instead of creating a completely new class.

## Factory providers

Another way to create a provider is using a factory provider.

The factory provider approach is ideal for situations when we need to create a provider, but the data it provides may change based on some data we don’t have until runtime.

A great example for when you may want to use one of these factory providers is if your application has some sort of authentication. So when a user is logged into the application, you need a provider to give you some data that is based on the user’s login credentials.

To do this, first we need to create a service that checks if someone can log in:

export class LoginService {  
       private allowUserIn: boolean;  
       canUserLogin(user: User) {  
           if(user.accessLevel === AccessStatus.AccessGranted) {  
    }   
}

This very simple service checks a user’s access level against an enum of access status. Next, we need to create a provider that will use this service:

const userAccessFactory = (userLogginIn: User) => {  
       return new LoginService(user)  
};

Here, we are creating a factory object that takes in a user object and returns our LoginService. The next stage is to set this factory and provider:

export let loginServiceProvider = { provide: LoginService, useFactory:userAccessFactory };

So, now we have LoginServiceProvider, which we can use in a provider array (which, as we already know, can be at the @Component level or the @NgModule level).

Factory providers allow users to build up a provider that is based on data we are loading through a service. You can see many uses for this approach, not just checking user-access levels. If our service loads data via an external API (something we will be exploring more later in this chapter), we could build a provider that gets the result of the data access service, concatenates this data, and sets it as a provider.

## Predefined tokens and providers

Angular itself gives us access to some providers we can use as developers to hook into when an application starts up. So, if you need to run a piece of code as the Angular framework starts up an application, we can do that through predefined tokens and providers.

There are three tokens available to us:

| **Token** | **Purpose** |
| --- | --- |
| **PLATFORM\_INITIALIZER** | A callback run when the Angular platform starts |
| **APP\_BOOTSTRAP\_LISTENER** | A callback for each Component that is used to Bootstrap the app; these components are set in app.module.ts |
| **APP\_INITIALIZER** | A callback for when the application is initialized |

So, we can create a provider that is run when these callbacks are triggered. In a provider array, we could create pass a factory that is run when one of these predefined stages runs:

[{  provide: PLATFORM\_INITIALIZER,   
    useFactory: platformInitializedFactory },  
    {provider: APP\_INITIALIZER,   
    useFactory: appInitializerFactory }]

Hopefully now you can see that providers give us many options for how we structure our applications and how we can use providers in so many different ways to supply various parts of our application with data they may need. How you use these providers, and which versions you use, depends on the business needs of your application.

# Services in Angular

Let's take a closer look at the services and the best practices for writing services in Angular.

**We'll cover the following**

* [The role of services in Angular](https://www.educative.io/courses/getting-started-with-angular/39rJG0pkEmM#The-role-of-services-in-Angular)
  + [Example: Check username and password](https://www.educative.io/courses/getting-started-with-angular/39rJG0pkEmM#Example:-Check-username-and-password)
* [Adding a service to a component](https://www.educative.io/courses/getting-started-with-angular/39rJG0pkEmM#Adding-a-service-to-a-component)
  + [Step 1: Add import statement](https://www.educative.io/courses/getting-started-with-angular/39rJG0pkEmM#Step-1:-Add-import-statement)
  + [Step 2: Make an instance of service](https://www.educative.io/courses/getting-started-with-angular/39rJG0pkEmM#Step-2:-Make-an-instance-of-service)
  + [Step 3: Call methods of our service](https://www.educative.io/courses/getting-started-with-angular/39rJG0pkEmM#Step-3:-Call-methods-of-our-service)
  + [Expected output](https://www.educative.io/courses/getting-started-with-angular/39rJG0pkEmM#Expected-output)
* [Public and private methods in services](https://www.educative.io/courses/getting-started-with-angular/39rJG0pkEmM#Public-and-private-methods-in-services)
  + [Private keyword](https://www.educative.io/courses/getting-started-with-angular/39rJG0pkEmM#Private-keyword)
  + [Public keyword](https://www.educative.io/courses/getting-started-with-angular/39rJG0pkEmM#Public-keyword)
  + [Using private and public functions in a service](https://www.educative.io/courses/getting-started-with-angular/39rJG0pkEmM#Using-private-and-public-functions-in-a-service)

After this discussion on providers, it may be hard to see what role services provide in Angular. If we’ve learned that providers can provide data to our components and modules, why would we need services?

## The role of services in Angular

Well, if providers give our application data and access to other modules and components, then services are **where our application’s business logic happens**.

**Services** make it easier to share the common functionality between the components.

The Angular framework doesn’t tell you that you have to use services, but the framework, through Dependency Injection, does make it extremely easy to write and consume services.

### Example: Check username and password

We could write all the business logic of an application at the @Component level. For example, a component could have a function that takes a username and password and makes an API to check that they are correct before allowing a user to log into the application.

This is fine, but what if we wanted to check the username and password somewhere else in the application? We would have to write the same code all over again in the new component that needs to check the login details.

Having a service that the components can delegate this logic to makes it easier to share common functionality between them, **removing the need for duplicate code that is hard to manage and maintain**

## Adding a service to a component

Now we know that the role of a service is to handle common functionality between components, let’s take a closer look at how to add this service to the component.

We know that a service needs to have the @Injectable operator in order for Dependency Injection to be aware of the class and know that it can be injected. Then there are two steps to adding the service in the component.

### Step 1: Add import statement

First, in the @Component file, we need to add an import statement to tell the component where the source of the service is. To do this, we add the import statement at the top of the TypeScript file of the component with all the other import statements:

import { ClientService } from '../client.service';

This tells the @Component class where to go to find the TypeScript class of the service.

### Step 2: Make an instance of service

Now that this import statement is in place, we can make an instance of the service to use throughout the component code. To do this, we create a local reference to the service in the constructor of the @Component class, like this:

constructor(private clientService: ClientService) {}

### Step 3: Call methods of our service

Now we have this new private property to use when we want to make a call to any of the public methods of this ClientService. For example, in an onInit lifecycle event of the component, we could call a method from the service to log that this component has been created, like this:

ngOnInit() {  
       this.clientService.log('Component has started');  
}

In this code example, the lifecycle hook (we know about these lifecycle events from the [Components, Templates and Forms chapter](https://www.educative.io/collection/page/10370001/4603693004488704/6414012458729472)). In this initialization lifecycle event, we call the log() method of our ClientService service using the local reference to this service we made in the constructor.

If we want to use another public method of the service, we would use the same local reference:

this.clientService.save(clientDetails);

Here, we are calling the save() public method of ClientService.

📝 **Note:** Press the **RUN** button to compile and serve the application. Once the app compiles, click on the URL given after Your app can be found at, and see what happens!

###### /

e2e

src

app

about

navigation

page-not-found

custom-material.module.ts

shared

company

clients

client.ts

client.service.ts

client.service.spec.ts

client.module.ts

client-form

client-page

client-page.component.html

client-page.component.scss

client-page.component.spec.ts

client-page.component.ts

search

app-routing.module.ts

app.component.html

app.component.scss

app.component.spec.ts

app.component.ts

app.module.ts

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.scss

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**client-page.component.ts**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

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

import { Client } from '../client';

import { ClientService } from '../client.service';

@Component({

  selector: 'app-client-page',

  templateUrl: './client-page.component.html',

  styleUrls: ['./client-page.component.scss'],

  providers: [ClientService]

})

export class ClientPageComponent implements OnInit {

  constructor(private clientService: ClientService) {}

  ngOnInit() {

       this.clientService.log('Component has started');

  }

  saveClient(clientDetails: Client) {

      console.log('clientDetails', clientDetails);

      this.clientService.save(clientDetails);

  }

}





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/>

Client Contacts Manager Application with ClientService

### Expected output

You should see the following output in the console:

Expected output: "Component has started"

## Public and private methods in services

We’ve mentioned this idea of a public method, but it’s worth explaining what we mean by a public method and how a TypeScript class, which a service is, can have both public and private methods.

In many programming languages, there is the concept of a class having both public and private methods and properties.

### Private keyword

The private keyword tells the TypeScript compiler that it’s not OK for anything outside of this class to access either the private property or the private method.

### Public keyword

Likewise, the public keyword tells the TypeScript compiler that it is OK for other classes (services, components, and so on) to access and use the properties and methods marked public.

### Using private and public functions in a service

When we are writing services for our Angular applications, it is worth thinking about when we want to use both the public and private keywords. Do we want all properties and methods to be public? Or do we want to make only a few properties and methods public and everything else private?

A good approach to this is to think of **creating a service with its own public API, in a way that other classes can access the functionality of the service**.

📝 **Note:** An **API** can be described as a set of functions and procedures allowing the creation of applications, that access the features or data of an operating system, application, or other services.

A **service needs to be a black box**, where other classes (the other services and components) do not need to know how a service works. All they care about is that they can make a call to the service and it gives them back what they need.

Service is like a black box

Why make our services black boxes to the other classes in our application? Well, one great benefit of this is it **allows us to make changes to the way a service is implemented without affecting the API into the Service class**. If we don’t change how a component of another service makes calls into a service, then the internal workings can be refactored without breaking existing code.

We’re starting to get into programming theory, which this course isn’t about, but it is worth spending a bit of time investigating best practices for writing services in order for you to know what the best approaches are when it comes to writing services for your Angular applications.

# A Look at the Services Angular Provides

Let's explore the different services that Angular provides.

**We'll cover the following**

* [Angular services](https://www.educative.io/courses/getting-started-with-angular/m7EP1DmNxPO#Angular-services)
* [Common set of services](https://www.educative.io/courses/getting-started-with-angular/m7EP1DmNxPO#Common-set-of-services)
* [common/http package](https://www.educative.io/courses/getting-started-with-angular/m7EP1DmNxPO#common/http-package)

## Angular services

Angular provides a number of services as part of the framework and as Angular developers, we can use these services in our applications. To see what services are available, we can go to the [official Angular docs](https://angular.io/api?type=class), where we can see all the services available to us.

Angular services

## Common set of services

As you can see on this page, the services are categorized by the type of functionality they offer. For example, we have a **common set of services**. These are services that offer functionality that is common in all Angular applications, such as using a **Location service** that provides the functionality to interact with the browser’s URL. There is a lot of functionality that comes as part of Angular, so it is worth looking through the API documentation to see what is available.

## common/http package

One area of the Angular API is the [common/http](https://angular.io/api/common/http) package. This contains all the functionality we may need to access external data through an API, which is a central part of building a modern web application.

# The HttpClient Service

Let's explore the HttpClient service, which we will use to make API calls in our Angular applications.

**We'll cover the following**

* [The HttpClient service](https://www.educative.io/courses/getting-started-with-angular/mEYA1jLNj2O#The-HttpClient-service)
* [The XMLHttpRequest Interface](https://www.educative.io/courses/getting-started-with-angular/mEYA1jLNj2O#The-XMLHttpRequest-Interface)
* [The fetch API](https://www.educative.io/courses/getting-started-with-angular/mEYA1jLNj2O#The-fetch-API)
* [The HttpClient service](https://www.educative.io/courses/getting-started-with-angular/mEYA1jLNj2O#The-HttpClient-service)
* [Adding the HttpClient service](https://www.educative.io/courses/getting-started-with-angular/mEYA1jLNj2O#Adding-the-HttpClient-service)
  + [Step 1: Update AppModule](https://www.educative.io/courses/getting-started-with-angular/mEYA1jLNj2O#Step-1:-Update-AppModule)
  + [Step 2: Update ClientService](https://www.educative.io/courses/getting-started-with-angular/mEYA1jLNj2O#Step-2:-Update-ClientService)

## The HttpClient service

In modern web applications, there are a number of ways to access an external API. As we are using Angular, we have to use what the browser gives us. Thankfully, modern browsers support two ways of reaching an API, which are as follows:

Accessing external APIs

## The XMLHttpRequest Interface

The **XMLHttpRequest interface** is an API that uses an object’s methods to transfer data from a web server.

These methods are known as the Request methods and they consist of the following:

* OPTIONS
* GET
* POST
* PUT
* DELETE
* HEAD
* TRACE
* CONNECT
* PATCH

Normally, we use the get, post, put and delete methods when accessing an API, but as you can see, the XMLHttpRequest interface has a few more methods that we could use.

## The fetch API

The fetch API is a new API that is available in modern browsers such as Chrome, Safari, Firefox, and Edge (not Internet Explorer).

**Fetch API** allows us to access APIs through a set of objects.

Each object has its own set of methods we can use to access data. The objects that the fetch API provides are as follows:

* WindowOrWorkerGlobalScope.fetch()
* Headers
* Request
* Response

The fetch API has been designed to look similar to the XMLHttpRequest interface so developers who have used this approach before will be familiar with how to implement the Fetch API.

## The HttpClient service

The HttpClient service uses XMLHttpInterface as its backbone, so we as Angular developers can use the HttpClient service in our applications, which in turn uses the browser’s support of XMLHttpInterface to access external APIs.

## Adding the HttpClient service

When we need to add the HttpClient service to one of the Angular applications, we first need to import HttpClientModule, in which the HttpClient service sits.

### Step 1: Update AppModule

To do this, we simply add HttpClientModule to our list of modules in the main app.module.ts file of our app. With this module is part of the main module, we can import the HttpClient service into one of our own services:

@NgModule({  
  declarations: [AppComponent, NavigationComponent],  
  imports: [  
    BrowserModule,  
    HttpClientModule,  
    AppRoutingModule,  
    BrowserAnimationsModule,  
    CustomMaterialModule,  
    ClientModule,  
    CompanyModule,  
    AboutModule,  
    SharedModule,  
  ],  
  providers: [ClientService],  
  bootstrap: [AppComponent],  
})  
export class AppModule {}

Without this HttpClientModule, if we try to add the HttpClient service the Angular compiler will throw an error.

### Step 2: Update ClientService

Next, we simply add HttpClient service to any of our own services where we wish to make API calls:

###### /

e2e

src

app

about

navigation

page-not-found

custom-material.module.ts

shared

company

clients

client.service.ts

client.service.spec.ts

client.module.ts

client-form

client-page

search

app-routing.module.ts

app.component.html

app.component.scss

app.component.spec.ts

app.component.ts

app.module.ts

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.scss

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**client.service.ts**

1

2

3

4

5

6

7

8

9

10

11

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

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

@Injectable({

  providedIn: 'root'

})

export class ClientService {

  constructor(private httpClient: HttpClient) {}

}





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/>

Client Contacts Manager Application with HttpClient

# Features of the HttpClient API

Let's explore how to make REST API calls using the HttpClient service of Angular to the backend of our application.

**We'll cover the following**

* [REST API calls](https://www.educative.io/courses/getting-started-with-angular/Bn86kZkNGjx#REST-API-calls)
* [Making a GET request](https://www.educative.io/courses/getting-started-with-angular/Bn86kZkNGjx#Making-a-GET-request)
  + [Implementation](https://www.educative.io/courses/getting-started-with-angular/Bn86kZkNGjx#Implementation)
  + [Observables and subscribing](https://www.educative.io/courses/getting-started-with-angular/Bn86kZkNGjx#Observables-and-subscribing)
* [Making a DELETE request](https://www.educative.io/courses/getting-started-with-angular/Bn86kZkNGjx#Making-a-DELETE-request)
  + [Implementation](https://www.educative.io/courses/getting-started-with-angular/Bn86kZkNGjx#Implementation)
* [Making a POST request](https://www.educative.io/courses/getting-started-with-angular/Bn86kZkNGjx#Making-a-POST-request)
  + [Implementation](https://www.educative.io/courses/getting-started-with-angular/Bn86kZkNGjx#Implementation)
* [Making a PUT request](https://www.educative.io/courses/getting-started-with-angular/Bn86kZkNGjx#Making-a-PUT-request)
  + [Implementation](https://www.educative.io/courses/getting-started-with-angular/Bn86kZkNGjx#Implementation)

## REST API calls

Now that we have added a HttpClient service to our application let’s take a look at the API of this class to determine what we can do before we go into the implementation.

Through the HttpClient class, we can make REST API calls to the backend of our application. These REST API calls are as follows:

REST API calls

These REST calls match the Request methods from the XMLHttpRequest object that modern browsers support. So, we can see that the HttpClient class is making use of the browser’s built-in API capabilities.

📝 **Note:** REST is an architectural structure for creating web services. When a web service follows the REST structure, it is known as a RESTful web service. The name **REST comes from its full name, Representational State Transfer**. A RESTful web service sends requests to systems so that it can access data from these systems (for example, a web server). For more information on REST, see the [Representational State Transfer](https://en.wikipedia.org/wiki/Representational_state_transfer) article on Wikipedia.

So, let’s look at how to make each of these REST calls using the HttpClient service and understand how they are implemented.

## Making a GET request

Making a GET request using the HttpClient service is very straightforward. Once we have injected the class into our service, we can write a function that uses the get() method to call an external API endpoint. Then, we subscribe to the Observable that the get() method returns. Finally, we listen to the observable, and it returns the response from the API call.

📝 **Note:** An **API endpoint** is a reference to a URL that accepts requests. These requests may or may not be REST requests. For a list of great publicly accessible APIs, check out this [GitHub repo](https://github.com/public-apis/public-apis).

### Implementation

That’s the theory. This is how it looks when we implement it. Let’s imagine we are building an Angular application that displays the latest train times, and, as part of this application, we have a page that returns all the current departure times. For this application, we may have a page component that shows all the latest departure times in a table.

As part of this component, we are using the Component lifecycle hook OnInit() to load the data for the page from an external API. Our OnInit() function would look something like this:

ngOnInit() {  
       this.trainTimesService.getDepartures().subscribe(response => {  
           this.departuresList = response  
       });  
}

In this ngOnInit() handler, there is a call to TrainTimesService using its getDepartures() method, which is where the HttpClient get function is used. Then we subscribe to the observable that HttpClient get() returns, and when we get the data back, we set a response to a local property called departuresList.

So, this is how we call the service in our component. Let’s look at the getDepartures() method in the TrainTimes service to see how we use the HttpClient get() method:

trainAPIUrl = 'https://api.trains.com/departures';  
   getDepartures()  {  
       return this.httpClient.get(this.trainAPIUrl);  
}

In this method, we are simply returning the observable that the httpClient.get method returns. When we make the get call, we’re passing in the URL of our API, which in this example is an API that returns a list of departures.

As you can see from these two snippets of example code, the HttpClient service is very powerful. There isn’t a lot of code here, and we are managing to make a call to an external API with just one line of code.

We’ve spoken about observables and subscribing to observables when looking at how to make HTTP Get calls, and we will be going into more depth on what Observables are in the next chapter, [Observables and RxJs](https://www.educative.io/collection/page/10370001/4603693004488704/4959879964917760), but it helps to go over the concepts while we are looking at the HttpClient service that makes use of Observables.

### Observables and subscribing

When we make an Observable using the HttpClient service, we are simply using an Observable object. This is an object that has certain properties and methods that allow it to open a connection to another data source (this data source could be an API or an event on a button). The Observable watches the data source, then when something happens on the data sources end, an API returns some data, and the observable reacts to this return of data and performs an action.

How does Observable know when something has happened? Well, it subscribes to this data source. It basically says, ‘I am interested in the data or action you return, so I’m going to listen for you to return the data and when I hear that, I’ll perform an action’.

So, in the component from the previous example, we subscribe/listen to the observable that the HttpClient get method provides, and when we hear the observable return the data, we respond.

Observable and subscribing

This approach is different from the previous version of Angular, where when we made API calls, we would get some data and then stop listening for data to be returned. Using an observable allows us to listen to when data is available so we can load data over time, instead of the previous approach, where loading data was a one-time action. With observables, we can listen for data to be returned over time and react when data is returned.

This use of observables in Angular is part of a more reactive programming approach that modern web applications use. We will be looking into this approach in the next chapter, but for now, we just need to understand that when we use an HttpClient method, it is returning an observable object we can subscribe to and listen to the response.

## Making a DELETE request

Again, making a delete request using the HttpClient service is very straightforward.

To make a Delete request, we follow the same pattern: have a method in our service that uses a HttpClient method and returns an observable. Then, at the component level, we subscribe to this observable, and when we subscribe, the method is called.

### Implementation

Going back to our fictional train service, a delete request would look like this:

trainAPIUrl = 'https://api.trains.com/ticket';  
  
deletePassengerTicket (ticketReference: number): Observable<{}>{  
       const url = `${this.trainAPIUrl}/${ticketReference}`;  
       return this.httpClient.delete(url, httpOptions)  
}

Here, we are taking in a ticket reference number as an identifier that the API needs to find the record to delete. We add this reference number to our trainAPIUrl, to make a new API path ( https:/​/​api.​trains.​com/​ticket/​21 ), then we call the delete method of HttpClient.

At this point, nothing has happened. It’s only when the deletePassengerTicket() method has been subscribed to that this code runs. So, again at the component level, we need to have a method that subscribes to the observable that this method returns.

So, we could have a deleteTicketRequest() method in our component like this:

deleteTicketRequest(ticketRef: number){   
    this.trainService.deletePassengerTicket(ticketRef).subscribe()  
}

This will call the deletePassengerTicket() method because we are subscribing to it. Even if the observable returns nothing, as in this example, we still need to call subscribe() to make it run.

## Making a POST request

After getting data, the second most common function we perform with APIs is posting data. This is where we send some data to an API endpoint, for example, posting some data from a form or sending an update to a record in a table. For this, we use the HttpClient service POST method.

### Implementation

The POST method is very similar to the GET method. Again, using our fictional train service, making a POST method call looks like this:

trainAPIUrl = 'https://api.trains.com/buyticket';  
     
buyTicket(ticket: TicketInformation)  {  
   return this.httpClient.post<TicketInformation>(this.trainAPIUrl,ticket, httpOptions);  
}

The POST method returns an Observable that we can subscribe to in the components buyCustomerTicket() method:

buyCustomerTicket(ticket: TicketInformation) {  
       this.trainService.buyTicket().subscribe((savedTicketInfo) => {  
           console.log(savedTicketInfo);  
       })  
}

A couple of things to notice here: the first is that in the post method call, we are setting the return type, <TicketInformation>, so that when the API returns some data, it is returned as an object with the properties of a TicketInformation object, which is an object we have defined in our code.

The second thing to notice is that we are passing in some httpOptions in our post method. These options are what the API is expecting when making an API call, for example, setting the headers for HTTP calls, which would look like this:

import { HtppHeaders } from '@angular/common/http';  
const httpOptions = {  
    headers: new HttpHeaders({ 'Content-Type': 'application/json',  
   'Authorization': 'authentication-token' })  
};

## Making a PUT request

The put request is slightly different from a post request. A put request will replace an item with the data that is passed as part of the put request. With a post, we’re sending some data to a backend via the API then the backend can decide what to do with the data it has received. post requests are used a lot more for data updates than a put request, but it is still a valid approach.

Again, we follow the common pattern for using the HttpClient service. We create a method in our service, but it’s not until the observable of that method is subscribed to that the put method will be called.

### Implementation

Here’s an example using our mock train service:

updatePassenger (passenger: Passenger): Observable<Passenger> {  
       return this.httpClient.put<Passenger>(this.trainsAPIURL, passenger, httpOptions) ;  
}

In this updatePassenger() method, we pass in an object that has a type of Passenger, so it has all the properties we expect a Passenger object to have. Then, when we subscribe to the observable this method the data it returns will be set to the same Passenger type because we have set it like this Observable<Passenger>.

Then at the Component level, we need to call this updatePassenger() method and subscribe to the observable it returns.

updateCurrentPassenger(currentPassenger: Passenger){  
    this.trainService.updatePassenger(currentPassenger).subscribe();  
}

Back

# Advanced Features of HttpClient Service

Let's explore how we can set the header options and pass parameters in an HTTP request, intercept HTTP requests and responses and listen to HTTP events.

**We'll cover the following**

* [Setting the HttpHeader options](https://www.educative.io/courses/getting-started-with-angular/Y58RJNrrKx2#Setting-the-HttpHeader-options)
* [Pass parameters using HttpParams](https://www.educative.io/courses/getting-started-with-angular/Y58RJNrrKx2#Pass-parameters-using-HttpParams)
* [Intercepting requests and responses](https://www.educative.io/courses/getting-started-with-angular/Y58RJNrrKx2#Intercepting-requests-and-responses)
  + [Advantages](https://www.educative.io/courses/getting-started-with-angular/Y58RJNrrKx2#Advantages)
* [Listening for HTTP events](https://www.educative.io/courses/getting-started-with-angular/Y58RJNrrKx2#Listening-for-HTTP-events)

As we have seen, the HttpClient service makes it easy to call a RESTful API, but there is more we can do with our HttpClient calls. We’re now going to look at some of the more advanced features to see how we can use them in our Angular applications.

## Setting the HttpHeader options

One of the first things we can do is set the header options just before we make an HTTP request.

For example, we may have a request that requires a different set of header options from the other HTTP requests we’re making in a service. Maybe when we are making a post request, we need to provide a different authentication token from the other HTTP requests we have in service.

To do this, we can use the set() method of the httpOptions object to create a new set of options. In our earlier example, we created httpOptions like this:

import { HttpHeaders } from '@angular/common/http';  
  
const httpOptions = {  
       headers: new HttpHeaders({ 'Content-Type': 'application/json', 'Authorization': 'authentication-token' })  
};

Here, we’ve created a const variable with HttpHeader options that can be used in all our HTTP requests, but if we want to make a new header option, we have to use the set() method to change the options:

httpOptions.headers = httpOptions.headers.set('Authorization', 'new-auth-token');

This ability to update the header options allows us to easily set them if we need to make a change. If your application uses a temporary authentication token for an initial request and is then returned a new token, we can set the new authentication token using this method.

## Pass parameters using HttpParams

Another useful class available to us in Angular is the HttpParams class. This allows us to create an object of all the parameters we may pass in an API call. Why would we want to do this? Well, one reason is we may want to URL-encode our parameters before passing them in a request.

📝 **Note:** **URL-encoding** means we convert the parameter to a format that can be passed as part of a URL. If we have parameters with characters in them that will break the URL of the request – special characters and so on – encoding them means these characters do not break the API URL.

To use the HttpParams class, we create an object that will contain all the parameters we have. This object is then passed as our options argument in our get or post requests:

const paramOptions = new HttpParams().set('orderBy','lastname').set('limitTo': '100');

In this example, we are creating a new set of parameter options, one called orderBy and one called limitTo, which would return some JSON from an API ordered by the last name and consisting of only 100 records.

You can see that we are chaining these parameters together one after the other. We could do this with a few more HttpParams in this object to build up a larger set of options. Eventually, they all will be passed to the API request as the paramOptions object.

## Intercepting requests and responses

Wouldn’t it be great to be able to intercept all the API requests that your application makes in order to manipulate the request and to ensure that all responses from the API request go through the same process? Well, you can by writing an interceptor class for your application.

To write an interceptor, we need to write a TypeScript class that implements the [HttpInterceptor](https://angular.io/api/common/http/HttpInterceptor" \t "_blank) interface.

This example shows how we can add a new header to all our requests:

@Injectable()  
export class AuthenticationInterceptor implements HttpInterceptor {  
       constructor(private authenticationService:AuthuenticationService) {}  
         
       intercept(req: HttpRequest<any>, next:HttpHandler):Observable<HttpEvent<any>> {  
           const authRequest = req.clone({  
               headers: req.headers.set('X-CustomAuthHeader',authenticationService.getToken())  
           });  
       return next.handle(authRequest);  
    }   
}

Now, with this service, all HttpRequests will go through the interceptor and have X-CustomAuthHeader set to the token the Authentication Service returns. The important thing to notice here is the intercept() function, which the class needs to provide because it is implementing the HttpInterceptor interface.

This function takes in two arguments: the request and the next handler. The request (called req in this example) is the HTTP request being made. It is an object that contains information about the request. As you can see from the example code, we’re setting the headers of the request being made. The next argument is an HttpHandler that needs to be called for the interceptor to move onto the next HTTP request in the queue. This is why at the end of the intercept() function, we call next.handle(authRequest).

To add this HTTP Interceptor class to our application, we set it as an HTTP\_INTERCEPTOR provider in the main app.module.ts:

@NgModule({  
   declarations: [ AppComponent ],  
   imports: [ BrowserModule, HttpClientModule ],  
   providers: [{   
       provide: HTTP\_INTERCEPTORS,   
       useClass: AuthenticationInterceptor,   
       multi: true } ],  
       bootstrap: [AppComponent]  
   })  
   export class AppModule { }

Now that this is set, the AuthenticationInterceptor class will be used. The great thing about the HttpClient class, as opposed to the original HTTP class from Angular 2, is that we can add multiple HTTP\_INTERCEPTORS to our application. We simply write a new class that implements the HttpInterceptor interface, then set it as a new provider in AppModule.

### Advantages

Other benefits of HttpInterceptor are that it can look for error codes returned from HTTP requests. Then, if it sees an error, the interceptor can redirect the user to an error page or update a log of errors.

We can also use HttpInterceptor to add a prefix to all HTTP requests if our API demands that all HTTP requests have a prefix on them, instead of having to remember to add this prefix on every HTTP request we make in our services. This prefix can be added via an HTTP Interceptor so that when a request is made, it goes through this interceptor, which adds the prefix.

## Listening for HTTP events

The last thing we are going to look at is HTTP events. The HttpClient service provides us with events that we can look out for as part of a request being made. With these events, we can do some clever things like informing the user of the status of a request as it’s being made. This would be especially helpful in larger HTTP requests, such as downloading a large file or a set of large images.

In this example, we are making a request, and in the subscribe() handler of the request, we are looking for the events in the request:

downloadFullReport() {  
    const apiUrl = '/company.com/reports/summer2018report';  
      
    this.httpClient.get(apiUrl,{},{ reportProgress:true }).subscribe( (event) => {  
        if(event.type === HttpEventType.DownloadProgress) {  
            console.log('Download progress is', event)  
        }  
          
        if(event.type === HttpEventType.Response) {   
            console.log('The response from the server is', event)  
        }   
    })  
}

A couple of things to note here. Firstly, we are passing the reportProgress argument in the request in order for the HttpClient service to provide these HTTP events. We are also using the [HttpEventType](https://angular.io/api/common/http/HttpEventType" \t "_blank) enum to check for the type of event. The use of an enum lookup removes the chance of spelling errors when looking for an event.

There are more than just two event types. We also have access to the following:

* Sent: Fired when the request is sent
* DownloadProgress: Download status event
* UploadProgress: Upload status event
* Response: The full response comes back as an event
* ResponseHeader: The response headers
* User: A custom event

This list of events can be really useful for our applications to monitor how our HTTP request is progressing. If a download progress event is taking too long, we can add some error handling to inform the end-user that there is an issue with the download, or we can tell them how much of the upload has been made.

# Creating Services for Client Contacts Manager Application

Let's create a service that contains In-Memory Web API to provide a to mak mock REST API calls to local memory and make use of services in order to save and manage data within the application.

**We'll cover the following**

* [Major changes](https://www.educative.io/courses/getting-started-with-angular/g7Z2j4pYODD#Major-changes)
* [app/services](https://www.educative.io/courses/getting-started-with-angular/g7Z2j4pYODD#app/services)
* [app/clients](https://www.educative.io/courses/getting-started-with-angular/g7Z2j4pYODD#app/clients)
  + [Create client.ts](https://www.educative.io/courses/getting-started-with-angular/g7Z2j4pYODD#Create-client.ts)
  + [Update client.module.ts](https://www.educative.io/courses/getting-started-with-angular/g7Z2j4pYODD#Update-client.module.ts)
  + [Create client.service.ts](https://www.educative.io/courses/getting-started-with-angular/g7Z2j4pYODD#Create-client.service.ts)
  + [Update client.module.ts](https://www.educative.io/courses/getting-started-with-angular/g7Z2j4pYODD#Update-client.module.ts)
  + [Update client-form](https://www.educative.io/courses/getting-started-with-angular/g7Z2j4pYODD#Update-client-form)
  + [Update client-page](https://www.educative.io/courses/getting-started-with-angular/g7Z2j4pYODD#Update-client-page)
  + [Update client.module.ts](https://www.educative.io/courses/getting-started-with-angular/g7Z2j4pYODD#Update-client.module.ts)
* [app/company](https://www.educative.io/courses/getting-started-with-angular/g7Z2j4pYODD#app/company)
* [Update app-routing.module.ts](https://www.educative.io/courses/getting-started-with-angular/g7Z2j4pYODD#Update-app-routing.module.ts)
* [Update custom-material.module.ts](https://www.educative.io/courses/getting-started-with-angular/g7Z2j4pYODD#Update-custom-material.module.ts)

## Major changes

In this version of the application, we have made the following changes:

* Added Services, one for the Client section and one for the Company section.
* Added forms for Adding, Editing, and Deleting both Clients and Companies.
* Used [angular-in-memory-web-api](https://github.com/angular/in-memory-web-api) to store data locally.
* Added lazy loading of routes.

## app/services

This contains a service, in-memory-data.service.ts, which uses a third-party library to provide make mock REST API calls to local memory. This allows the user to build a REST API to write data without having to use a hosting service. This service will be used by both the Client and Company Services.

import { Injectable } from '@angular/core';  
import { InMemoryDbService } from 'angular-in-memory-web-api';  
import { Client } from '../clients/client';  
  
@Injectable({  
  providedIn: 'root'  
})  
export class InMemoryDataService implements InMemoryDbService {  
  createDb() {  
   const clients =[{id:2,firstname:'Jane',lastname:'Doe',email:'jane.doe@example.com',telephoneNumber:'0777334455',companyName:'Example inc'},];  
  
   const company =[{id:1,companyName:'Google Inc',address1:'Googleplex',address2:'Mountain View',town:'San Fransisco',city:'San Fransisco',postcode:'SF123',country:'US',contactEmail:'info@google.com',staffCount:4,industry:'Tech'},];  
  
    return {clients, company};  
  }  
}

## app/clients

Let’s explore the files within the clients folder.

### Create client.ts

This file provides a class that is used as a data model in the Service calls.

export class Client {  
  id?: string;  
  firstname: string;  
  lastname: string;  
  email: string;  
  telephoneNumber: string;  
  companyName: string;  
}

### Update client.module.ts

We will also add the angular-in-memory-web-api service to app.module.ts file like this:

import { HttpClientInMemoryWebApiModule } from'angular-in-memory-web-api';  
import { InMemoryDataService} from './services/in-memory-data.service';  
  
@NgModule({  
    imports: [HttpClientInMemoryWebApiModule.forRoot(InMemoryDataService),]  
})

### Create client.service.ts

This service provides a way to make API calls to the mock API service we’re using in this example app, to save the details of a client, to get the details of a client by their ID number, to get a list of the clients in the system, and to delete a client from the system. The aim of this chapter is to introduce the use of Services So this service aims to show the different ways a service can be used.

### Update client.module.ts

Also, we will add the ClientService to the providers array like this:

import { ClientService } from './client.service';  
  
@NgModule({  
  providers: [ClientService],  
})

### Update client-form

In this file, I’ve added the ability to create a Client object, which is then sent via an event to a client-page component. This then uses the ClientService to save the new client’s details to the system. This change shows how to load a Service and how to make calls to Services.

### Update client-page

In this component, I use the ClientService to subscribe to a list of the clients in the system and handle when a new client is added to the system using the save() method of the ClientService. It shows examples of how to use a service.

### Update client.module.ts

Here, we add the ClientService to the application.

## app/company

These company files work very similarly to the client section. They provide a way for the user to search and see a list of the companies in the system. They can then select a company and see the full details of that company. They can either view, edit or delete/archive the company. Again, this uses the parent/child component approach.

* company-detail
* company-details-page
* company-form
* company-list
* company-page
* company-search-page
* company.service.ts
* company.ts
* company.module.ts

## Update app-routing.module.ts

We added new routes for adding new clients, for editing an existing client, and for searching for a client. Similarly, we added new routes for adding new companies, for editing an existing company, and for searching for a company.

## Update custom-material.module.ts

Also, we will add the following modules to the custom-material.module.ts file:

import { MatTableModule } from '@angular/material/table';  
import { MatListModule } from '@angular/material/list';  
import { MatIconModule } from '@angular/material/icon';  
  
@NgModule({  
  imports: [  
    MatTableModule,  
    MatListModule,  
    MatIconModule  
  ],  
  exports: [  
    MatTableModule,  
    MatListModule,  
    MatIconModule  
  ]  
})

📝 **Note:** We have also made changes in the navigation and search component.

###### /

e2e

src

app

about

clients

client-detail

client-details-page

client-edit-page

client-form

client-list

client-page

client-search-page

client.module.ts

client.service.spec.ts

client.service.ts

client.ts

company

navigation

page-not-found

search

services

shared

app-routing.module.ts

app.component.html

app.component.scss

app.component.spec.ts

app.component.ts

app.module.ts

custom-material.module.ts

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.scss

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**client.module.ts**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

import { CommonModule } from '@angular/common';

import { CUSTOM\_ELEMENTS\_SCHEMA, NgModule } from '@angular/core';

import { FormsModule, ReactiveFormsModule } from '@angular/forms';

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

import { CustomMaterialModule } from '../custom-material.module';

import { SharedModule } from '../shared/shared.module';

import { ClientDetailComponent } from './client-detail/client-detail.component';

import { ClientDetailsPageComponent } from './client-details-page/client-details-page.component';

import { ClientEditPageComponent } from './client-edit-page/client-edit-page.component';

import { ClientFormComponent } from './client-form/client-form.component';

import { ClientItemComponent } from './client-list/client-item/client-item.component';

import { ClientListComponent } from './client-list/client-list.component';

import { ClientPageComponent } from './client-page/client-page.component';

import { ClientSearchPageComponent } from './client-search-page/client-search-page.component';

import { ClientService } from './client.service';

const routes: Routes = [

  {

    path: 'clients/new',

    component: ClientPageComponent

  },

  {

    path: 'clients/search',

    component: ClientSearchPageComponent

  },

  {

    path: 'clients/details/:id',

    component: ClientDetailsPageComponent

  },

  {

    path: 'clients/edit/:id',

    component: ClientEditPageComponent





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/>

Client Contacts Manager Application with Services

# Observables and RxJS

Let's learn about Observables and RxJS in this chapter.

**We'll cover the following**

* [Chapter learning outcomes](https://www.educative.io/courses/getting-started-with-angular/BnBgZNWP4Ok#Chapter-learning-outcomes)
* [Client contacts manager application](https://www.educative.io/courses/getting-started-with-angular/BnBgZNWP4Ok#Client-contacts-manager-application)

## Chapter learning outcomes

Now that we know what services are and how to create them, we are going to move on to looking at Observables. We are going to learn **what an Observable is, what role it provides, and what makes an Observable object**. Once we have discussed the theory of Observables, we will move on to looking at the **RxJs library** that has been added to Angular. We will look at what RxJs is and what role it provides to Angular, as well as how we can **add features of RxJs to our services** to make them more resilient and error-proof.

We’ve spoken a few times already about Observables and RxJs but never looked into them in great depth, including how to use them and why we use them in Angular. In this chapter, we will explore both topics in more detail.

In this chapter, we will cover the following topics:

* Observable objects
* How to use the Observable pattern in Angular
* How to create Observables
* How to use Observables
* What RxJs is and an introduction to Reactive Programming
* How we can use the power of RxJs in our Angular applications

There is a lot of theory to go through and a lot of concepts we need to learn and understand. By the end of this chapter, you will see the benefits that this approach brings to our Angular applications.

## Client contacts manager application

📝 **Note:** Press the **RUN** button to compile and serve the application. Once the app compiles, you can see its output in the **Output** tab or by clicking on the URL given after Your app can be found at.

###### /

e2e

src

app

about

clients

client-detail

client-detail.component.html

client-detail.component.scss

client-detail.component.spec.ts

client-detail.component.ts

client-details-page

client-edit-page

client-form

client-list

client-page

client-search-page

client.module.ts

client.service.spec.ts

client.service.ts

client.ts

company

navigation

page-not-found

search

services

shared

app-routing.module.ts

app.component.html

app.component.scss

app.component.spec.ts

app.component.ts

app.module.ts

custom-material.module.ts

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.scss

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**client-detail.component.html**

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

    <p>Telephone: {{ client?.telephoneNumber }}</p>

  </mat-card-content>

  <mat-card-footer>

    <button (click)="edit()" mat-raised-button color="primary">Edit</button>

    <button (click)="searchAll()" mat-raised-button color="primary">Search All Clients</button>

    <button (click)="delete(client)" mat-raised-button color="warn">Delete</button>

  </mat-card-footer>

</mat-card>

<mat-card \*ngIf="editView">

  <form [formGroup]="clientEditForm">

  <mat-card-header>

    <mat-form-field>

      <input matInput type="text" name="firstname" formControlName="firstname" />

    </mat-form-field>

    <mat-form-field>

      <input matInput type="text" name="lastname" formControlName="lastname" />

    </mat-form-field>

  </mat-card-header>

  <mat-card-subtitle>

    <h2>{{ client?.companyName }}</h2>

  </mat-card-subtitle>

  <mat-card-content>

    <mat-form-field>

      <input matInput type="text" name="email" formControlName="email" />

    </mat-form-field>

    <mat-form-field>

      <input matInput type="text" name="telephone" formControlName="telephoneNumber" />

    </mat-form-field>

  </mat-card-content>

  <mat-card-footer>

    <button mat-button mat-raised-button color="primary" (click)="save()">Save</button>





Run

Save

Reset

**Your app can be found at:**[https://ed-6434592358924288.educative.run](https://ed-6434592358924288.educative.run/)

Client Contacts Manager Application with RxJS

In this version of the application, we have made the following changes:

* Updated both the Client and Company services to show that Observables are being returned from the HttpClient calls.
* Updated both the Client Edit form and the Company Edit form to Reactive forms.
* Used RxJs in the Search Form Component, so now it starts searching as you type.
* Added OnChanges lifecycle hook to both the Client and Company edit forms in order to populate a Reactive form with the data passed in from the parent Component.



# Terminologies Related to RxJS

Let's understand important terminology related to reactive programming, e.g., Observable, Observer, Subscription, Operators, Subjects, Schedulers.

**We'll cover the following**

* [Observable and Observer](https://www.educative.io/courses/getting-started-with-angular/g218qlvz8K9#Observable-and-Observer)
* [Subscription](https://www.educative.io/courses/getting-started-with-angular/g218qlvz8K9#Subscription)
* [Subjects](https://www.educative.io/courses/getting-started-with-angular/g218qlvz8K9#Subjects)
* [Operators](https://www.educative.io/courses/getting-started-with-angular/g218qlvz8K9#Operators)
* [Schedulers](https://www.educative.io/courses/getting-started-with-angular/g218qlvz8K9#Schedulers)

When discussing Observables, RxJs, and how they are used in Angular, we need to understand some terminology that will repeatedly come up as we discuss these topics. These common terms are as follows:

## Observable and Observer

An **Observable** is an object that provides support for sending messages from publishers to subscribers within an application.

That’s the official explanation of what an Observable is. This means that when we create an object that uses the Observable pattern, this object has the ability to send out messages. These are messages that other objects that follow the Subscribe pattern can listen for, and when they hear these messages, they can act on them.

We had already seen examples of this in the [previous chapter](https://www.educative.io/collection/page/10370001/4603693004488704/5073909983477760#making-a-get-request) when we looked at services. There were many times where we set up a function in a service that returned an Observable object. This Observable object was then subscribed to at the component level, where we could act on the message (or response) that was sent from the Observable object coming from the service.

In a nutshell, Observable is the name we give an object that sends out the messages that other objects listen for.

The **Observer** is an object that subscribes or is interested in hearing the messages that an Observable is sending out.

## Subscription

A Subscription is a mechanism that the Observer uses to show that it is interested in these messages from the Observable. Think of it this way – if you create an object that is a type of Observable, then you create an object that is an Observer. The Observer subscribes to anything that the Observable sends out.

## Subjects

Subjects are similar to Observables, but they can send out messages to multiple Observers. They have been described as being similar to an EventEmitter in Angular, where the Subject sends out a message or value to multiple listeners. Like Operators, they are a part of RxJs but based on the Subject design pattern.

## Operators

Operators are a concept from RxJs and are methods that can be attached to an Observable. These methods perform operations on the Observable, and it returns a new Observable object that is still subscribed to by Observers. We will look at Operators later when we discuss RxJs, but for now, just know that they are methods from the RxJs library that we can use to perform operations on the Observables we create.

## Schedulers

Schedulers are also a part of RxJs, and we can use them to schedule when something happens within our application. For example, we may want to take a response from an Observable and manipulate the data that’s returned. Then, we may want to get some more data from another Observable, but not before the first Observable has finished its work. With Schedulers, we can set up a pipeline of work, which is powerful and shows how useful RxJS is to our Angular application.

# The Observable Object

Let's learn about an Observable object, its characteristics, and the design patterns that are used to define it.

**We'll cover the following**

* [The Observable object](https://www.educative.io/courses/getting-started-with-angular/N77pxOl1qw8#The-Observable-object)
  + [The next( ) handler](https://www.educative.io/courses/getting-started-with-angular/N77pxOl1qw8#The-next(-)-handler)
  + [The complete( ) handler](https://www.educative.io/courses/getting-started-with-angular/N77pxOl1qw8#The-complete(-)-handler)
  + [The error( ) handler](https://www.educative.io/courses/getting-started-with-angular/N77pxOl1qw8#The-error(-)-handler)
  + [Example: Relationship between the Observable and the Observer](https://www.educative.io/courses/getting-started-with-angular/N77pxOl1qw8#Example:-Relationship-between-the-Observable-and-the-Observer)
* [Example 1: Run an event without a reactive approach](https://www.educative.io/courses/getting-started-with-angular/N77pxOl1qw8#Example-1:-Run-an-event-without-a-reactive-approach)
* [Example 2: Run an event with a reactive approach](https://www.educative.io/courses/getting-started-with-angular/N77pxOl1qw8#Example-2:-Run-an-event-with-a-reactive-approach)
* [Comparison between the reactive and non-reactive approach](https://www.educative.io/courses/getting-started-with-angular/N77pxOl1qw8#Comparison-between-the-reactive-and-non-reactive-approach)
  + [Non-reactive approach](https://www.educative.io/courses/getting-started-with-angular/N77pxOl1qw8#Non-reactive-approach)
  + [Reactive approach](https://www.educative.io/courses/getting-started-with-angular/N77pxOl1qw8#Reactive-approach)
* [Observable in layman’s terms](https://www.educative.io/courses/getting-started-with-angular/N77pxOl1qw8#Observable-in-layman%E2%80%99s-terms)
* [The structure of an Observable object](https://www.educative.io/courses/getting-started-with-angular/N77pxOl1qw8#The-structure-of-an-Observable-object)
  + [The next( ) handler](https://www.educative.io/courses/getting-started-with-angular/N77pxOl1qw8#The-next(-)-handler)
  + [The error( ) handler](https://www.educative.io/courses/getting-started-with-angular/N77pxOl1qw8#The-error(-)-handler)
  + [The complete( ) handler](https://www.educative.io/courses/getting-started-with-angular/N77pxOl1qw8#The-complete(-)-handler)

## The Observable object

When an Observable object is created, we register an Observer object with the Observable. Then, as the Observable receives items, it calls all its subscribed Observer’s methods. There are three handler methods that the Observable can call:

Subscribed Observer's method

### The next( ) handler

The next() method sends a value to an Observer. A value can be anything a number, a string, an object, etc.

### The complete( ) handler

The complete() handler is fired when the Observable has no more items to send to its subscribed Observers.

### The error( ) handler

The error() handler is called by the Observable if there is a problem when it is emitting data to the subscribed Observers.

Observable object

These items that the Observables emit can be varied from a single click event, to a block of JSON data, to a series of messages from a backend push service. The Observable makes a non-blocking connection to its subscribed Observers, which allows items to be sent out over time to the Observer, each time calling the Observer’s next handler until there is either an error or all the items have been sent. It doesn’t matter what the items being sent are – the approach is the same. We create an Observable, register the Observers, and it’s not until the Observer’s Subscribe method is called that the items the Observable is loading will be called. Then, the Observable keeps calling the Observer’s next() handler until all the items have been pushed out by the Observable.

### Example: Relationship between the Observable and the Observer

The following code should make the relationship between the Observable and the Observer clearer:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

const { Observable } = require("rxjs");

// // Creating an Observable and passing an Observer object to its subscribe method

const observableObj = new Observable(

  (subscribe = (subscribedObserver) => {

    try {

      // Calling the next handler on each item until there are no more

      subscribedObserver.next(1);

      subscribedObserver.next(2);

      subscribedObserver.next(3);

      subscribedObserver.complete();

    } catch {

      // Calls the error handler if there is an issue

      subscribedObserver.error(error);

    }

  })

);

observableObj.subscribe((val) => console.log(val));





Run

Save

Reset

Subscribing to Observable

We are creating an Observable observableObj (line 4) and then calling its subscribe method (line 19). After that, we keep calling the next() handler on each item until there are no more (line 8-10). After that, the complete() handler is fired when there are no more items to send to the subscribed Observer (line 11).

The Observable is managing everything the Observer reacts to when it receives data.

This approach is different from earlier versions of Angular (AngularJS), where we would make a request for data, then wait for the data to be returned before moving on. Through the use of RxJS and Observables, our applications are more reactive to changes. Let’s look at a simple example to show you the difference between taking a reactive approach compared to the previous approach.

## Example 1: Run an event without a reactive approach

Say we have a page that has a button, and we want to run an event when the button is clicked. Previously, we would add a click event like this:

📝 **Note:** Press the **Run** button. You will see the button in the output. Press it and see what happens!

* JavaScript

1

2

3

4

5

6

7

const myButton = document.getElementById("myButton");

myButton.addEventListener("click", buttonClickHandler);

function buttonClickHandler() {

  myButton.textContent = "Hello World!";

}





Run

Save

Reset

Clicking the button will run an event, and the button text will be changed from Click Me! to Hello World!

## Example 2: Run an event with a reactive approach

Now, by taking a more reactive approach, we can do the same:

📝 **Note:** Press the **Run** button; you will see the button in the output, press the button and see what happens!

* JavaScript

1

2

3

4

5

6

const myButton = document.getElementById("myButton");

rxjs.fromEvent(myButton, "click").subscribe(buttonClickHandler);

function buttonClickHandler() {

  myButton.textContent = "Hello World!";

}





Run

Save

Reset

## Comparison between the reactive and non-reactive approach

Both examples are very similar. They both find an element on the page and handle when the button is clicked.

### Non-reactive approach

The first version adds an Event handler to the button for a Click event. When the button is clicked, the event listener calls buttonClickHandler(). We have to find the element, then register a click event listener to call a handler. The first approach has three steps:

* Find the element
* Add an event listener
* Register a handler.

### Reactive approach

In the second version, we are simply finding the element and subscribing to the click event of that element using the fromEvent operator of RxJs. **The second approach produces cleaner code** since only two things are happening:

* Finding the element
* Subscribing to the click event.

You can see that the second approach is just creating a connection and waiting for the element’s click event to be fired. When that happens, the Subscribe method of the Observer from that fromEvent reacts and calls the buttonClickHandler() method.

These examples show the subtle differences from what we used to do compared to what we can do now through Observables. It also shows the shift in mindset we need to have when working with Observables from the idea that we need to set up a load of Event Listeners to handle how our application works, to just Subscribing to events and then letting the application react to changes as the end-user performs tasks in the application.

## Observable in layman’s terms

Before diving into a large technical explanation of how an Observable is structured, let’s try and explain in layman’s terms what an Observable does.

An **Observable** is a way of creating a stream of data. This data is passed along this stream from the Observable to the Observer…

If you think of a **conveyor belt** that passes along boxes, this conveyor belt is only started when someone at the end of the belt switches it on. Then, the belt starts sending down boxes to the person who switched the belt on.

In this metaphor, an Observable is the conveyor belt, that is, a mechanism for passing along boxes (data), and the Observer is the person who starts the belt (the data stream). They switch on this belt by subscribing to the data stream. With this metaphor in mind, let’s start looking into the structure of an Observer.

Observable as a conveyer belt

## The structure of an Observable object

To create an Observer object, we need to have an object that implements the Observer interface, meaning that the object needs to implement methods for the three notifications that the Observables send out. These methods are, as we have already stated, as follows:

* next
* complete
* error

For an object to be an Observer, it must at least have an implementation for the next() method. The complete() and error() methods are optional, though making use of an error() handler is good practice.

### The next( ) handler

If an Observable sends out just one item, say, as part of a click event, then the next() handler may not be called because there is nothing next() for the handler to work with. There is just the one item, which in this instance is a click event. If, as we saw in the previous example, the Observable is sending out a series of three items (1,2,3), the next() handler will be called after each item to handle getting the next item. The next() handler invocation depends on how many items the Observable is emitting.

### The error( ) handler

The error() handler method is called if the stream of items from the Observable is interrupted by an issue, for example, if a series of messages being sent from an API is stopped due to an error being returned by the API. The error() handler is invoked when this happens. If there is no error handler, the errors are not captured, and the Observable silently fails without a way of seeing what the issue is.

### The complete( ) handler

The complete handler is called when the Observable notifies the subscribed Observers that it has completed sending out items. Then, the Observer can use this handler to inform the user that the data has completed loading or that there are no more messages being sent at this time.

It’s important to remember that an Observer object can define any of these notification handlers. But if the Observer doesn’t have a handler for these notification types, the notification is just ignored, which could lead to errors not being discovered if you don’t have a handler for the error notifications.

# Subscribing and Unsubscribing from Observables

Let's explore how we can subscribe and unsubscribe to the observables.

**We'll cover the following**

* [Subscribing to Observables](https://www.educative.io/courses/getting-started-with-angular/g7ZYVJzmopZ#Subscribing-to-Observables)
  + [HttpClient’s get() method](https://www.educative.io/courses/getting-started-with-angular/g7ZYVJzmopZ#HttpClient%E2%80%99s-get()-method)
* [Why unsubscribe from Observables?](https://www.educative.io/courses/getting-started-with-angular/g7ZYVJzmopZ#Why-unsubscribe-from-Observables?)
  + [Improve the performance of the application](https://www.educative.io/courses/getting-started-with-angular/g7ZYVJzmopZ#Improve-the-performance-of-the-application)
  + [Stop memory leakage](https://www.educative.io/courses/getting-started-with-angular/g7ZYVJzmopZ#Stop-memory-leakage)
* [Methods for unsubscribing from Observable](https://www.educative.io/courses/getting-started-with-angular/g7ZYVJzmopZ#Methods-for-unsubscribing-from-Observable)
  + [Calling the unsubscribe() method](https://www.educative.io/courses/getting-started-with-angular/g7ZYVJzmopZ#Calling-the-unsubscribe()-method)
    - [onInit life cycle](https://www.educative.io/courses/getting-started-with-angular/g7ZYVJzmopZ#onInit-life-cycle)
    - [onDestroy life cycle](https://www.educative.io/courses/getting-started-with-angular/g7ZYVJzmopZ#onDestroy-life-cycle)
  + [Using the takeUntil() operator from RxJs](https://www.educative.io/courses/getting-started-with-angular/g7ZYVJzmopZ#Using-the-takeUntil()-operator-from-RxJs)
  + [Using AsyncPipe](https://www.educative.io/courses/getting-started-with-angular/g7ZYVJzmopZ#Using-AsyncPipe)

## Subscribing to Observables

As we saw in the [Dependency Injection, Services, and HttpClient chapter](https://www.educative.io/collection/page/10370001/4603693004488704/5073909983477760), when we implemented one of the HttpClient methods, it returned an Observable, but it wasn’t until we called the Subscribe method of the Observable that it actually ran.

### HttpClient’s get() method

When we made a request to get some data using the HttpClient’s get() method, the data wasn’t loaded until we called the Subscribe method at the component level. We set up the HttpClient get() method call in our service and returned the Observable that the get() method returns. Then, in the component, we called the Subscribe of that returned Observable.

In this Subscribe method, we pass in the Observer. In this example, we are returning the Observable that the get() method creates for us:

const externalAPI = 'example.com/data';  
loadSomeData() {  
   return this.httpClient.get(externalAPI)  
}

Then in the Component, we call the Subscribe method of this Observable:

this.service.loadSomeData().subscribe((data) => {  
       console.log(data);  
})

This is a common pattern to use when working with Observables, but in the preceding code, we’re not defining this “subscribe” method. We’re just calling it. How is that possible? Well, we know that the get() method of the HttpClient returns an Object, which is an Observable. If we look at the official API documentation, we can see that it returns an [Observable](https://angular.io/api/common/http/HttpClient#get), and so by being an Observable, it must have a Subscribe method that we can call.

Observable and subscribing

## Why unsubscribe from Observables?

As well as subscribing to Observables, we can also unsubscribe from them, but why do we need to do that?

### Improve the performance of the application

The reason is that we can improve the performance of our application. If you think about it, each Observable we subscribe to is like creating an open connection that is never closed. As items are passed from the Observable to the Observer, the next() and, if defined, complete() methods are called. But they don’t close the connection.

Improve performance

If the Observable sends through four different items (messages from a backend server, for example), after each of these four messages, the next() handler is invoked. When the last of the four messages has been sent, the Observable calls the Observer’s complete() handler. However, the connection is still open, so if the Observable has another five messages to send, they can be sent to the Observer.

### Stop memory leakage

If an application uses Observables throughout the application without unsubscribing from them, there would be loads of open connections. This could lead to a memory leak in our application. Therefore, we need to unsubscribe from them in order to stop a memory leak from our application.

Prevent memory leakage

## Methods for unsubscribing from Observable

There are a couple of ways we can unsubscribe from the connection the Observables make, as follows:

* Calling the unsubscribe() method
* Using the takeUntil() operator from RxJs
* Using AsyncPipe

### Calling the unsubscribe() method

Let’s look at an example of using the unsubscribe() method:

export class DemoComponent implement OnInit, OnDestroy {  
  service: DemoService;  
  subscription: Subscription;  
  
  ngOnInt() {  
    this.subscription = this.service.getSomeData().subscribe((data) => {  
      console.log(data);  
    });  
  }  
  
  ngOnDestroy() {  
    this.subscription.unsubscribe();  
  }  
}

Here, we’re using two lifecycle hooks (we discussed lifecycle hooks in the [Components, Templates, and Forms chapter](https://www.educative.io/collection/page/10370001/4603693004488704/6414012458729472)): onInit and onDestroy.

#### onInit**life cycle**

In the onInit handler, we make a call to the service and get some data. This returns an Observable because, as we will remember from looking at the HttpClient service, the get() method returns an Observable. We then set this Observable to a Subscription object, which is from RxJs. The reason we use this Subscription object is that this object has an unsubscribe() method.

#### onDestroy**life cycle**

In the onDestroy life cycle hook handler (which is run when a component is destroyed), we call the unsubscribe() method of the Subscription Object, which unhooks the connection that’s made to the Observable we set to the Subscription object, which in this case is the Observable that’s returned from the get() method.

### Using the takeUntil() operator from RxJs

Calling the unsubscribe() method is one approach. However, we can also use the takeUntil() operator that RxJs provides:

export class DemoComponent implements OnInit, OnDestroy {  
  service: DemoService;  
  unsubscribe$: Subject;  
  ngOnInit() {  
    this.service  
      .getSomeData()  
      .takeUntil(this.unsubscribe$)  
      .subscribe((data) => {  
        console.log(data);  
      });  
  }  
  
  ngOnDestroy() {  
    this.unsubscribe$.next();  
    this.unsubscribe$.complete();  
  }  
}

Here, we are still using the life cycle hooks, but we are also using the takeUntil() operator of RxJs (we will go into the details of RxJs [later](https://www.educative.io/collection/page/10370001/4603693004488704/5720580392222720)). This takeUntil() operator keeps using a source Observable until the notifier Observable that we pass into the takeUntil() operator tells it to stop.

At the top of this demo component class, we create a new Observable called unsubscribe$ (the dollar sign is part of a naming convention that’s used when working with Observables). This Subject Observable is passed into the takeUntil() operator. Then, when it’s in the ngOnDestroy handler, the next() and complete() handlers of the unsubscribe$ Observable are called, which stops/unsubscribes the Observable being returned from the getSomeData() method. Basically, we are creating a new Observable that manages the first one.

### Using AsyncPipe

The third method we’re going to look at is using the AsyncPipe. This approach uses RxJs but with Angular templates. Let’s have a look at an example:

export class DemoComponent implements OnInit {  
  dataSubscription: Observable<SomeData>;  
  ngOnInit() {  
    this.dataSubscription = this.service.getSomeData();  
  }  
}  
  
     
// In our HTML template file   
<span>  
  {{ dataSubscription | async }}  
</span>

Here, we have our demo component, which has a local property called dataSubscription. This is an Observable that we expose to the template of this component. When we use this property in the template, we are also using the async pipe.

📝 **Note:** Pipes in Angular allow us to transform data into the desired output. There are many different pipes in Angular – ones that change the text to uppercase, ones that transform dates, and so on. They are very useful and worth exploring in the [official documentation](https://angular.io/docs).

The async pipe handles subscribing and unsubscribing for us. When this DemoComponent is destroyed, and the onDestroy() lifecycle hook that is part of all component is called (even if we don’t override it with our own version of the onDestroy hook), the async pipe will unsubscribe from the Observable.

As we can see, there are a variety of ways to unsubscribe from an Observable. What these three examples show is that libraries like RxJs help us manage Observables in our Angular applications. As we know, using Observables provides many advantages, but they do need to be managed. Using the methods we’ve just looked at, we can make use of Observables and still keep our applications performing well.

# Error Handling in Observables

Let's explore how we can handle errors in the subscribe method of the Observable.

**We'll cover the following**

* [Error handling](https://www.educative.io/courses/getting-started-with-angular/3jP6BkmENrr#Error-handling)
  + [Example](https://www.educative.io/courses/getting-started-with-angular/3jP6BkmENrr#Example)

## Error handling

Displaying errors when working with Observables is handled by the error() handler of the Observer object. As we know, an Observer has three types of handlers: next(), complete(), and error().

In the **error() handler**, we can see if any problems have occurred.

### Example

Let’s look at an example:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

const { Observable } = require("rxjs");

const observableObj = new Observable(

  (subscribe = (subscribedObserver) => {

    try {

      // calling the next handler on each item until there are no more

      subscribedObserver.next(1);

      subscribedObserver.next(2);

      // Error: write nxt instead of next

      subscribedObserver.nxt(3);

      subscribedObserver.next(4);

      throw new Error('Custom Error')

      subscribedObserver.complete();

    } catch (error) {

      // calls the error handler if there is an issue

      subscribedObserver.error(error);

    }

  })

);

observableObj.subscribe({

  next(data) {

    console.log("Here is some data:" + data);

  },

  error(err) {

    console.error("There is an error: " + err);

  }

});





Run

Save

Reset

Error handling in Observable

In this example, we have an Observable, which is being subscribed to. In the Subscribe method, we are passing in an Object, which is our Observer. This has two handler functions: one for next() and one for error(). The subscribed observer will receive 1 and 2 in the output but on line 10 it will catch an error, call the error() handler which will display the error, and the program will stop working.

If this Observable was returning data from an API and there was an issue getting this data, the error() function of the Observer would be invoked, displaying the error message in the console.error() method.

It is important to know that if the error() handler is called, then the Observable will stop producing data. The only time the Observable keeps producing a stream of data is when the next() handler is invoked. If either the complete() or the error() handlers are invoked, then the Observable will stop retrieving data.

📝 **Note:** This is important to know because if we don’t have an error handler and there is an error that we may not be aware of, our Observable will stop retrieving data.

# Hot and Cold Observables

Let's discuss how we can handle error and multicast operations in Observables.

**We'll cover the following**

* [Hot Observables](https://www.educative.io/courses/getting-started-with-angular/YVW5RQrDj5A#Hot-Observables)
  + [Example: click event](https://www.educative.io/courses/getting-started-with-angular/YVW5RQrDj5A#Example:-click-event)
* [Cold Observables](https://www.educative.io/courses/getting-started-with-angular/YVW5RQrDj5A#Cold-Observables)
  + [Example: Observable created using from operator](https://www.educative.io/courses/getting-started-with-angular/YVW5RQrDj5A#Example:-Observable-created-using-from-operator)
* [Multicasting in hot Observables](https://www.educative.io/courses/getting-started-with-angular/YVW5RQrDj5A#Multicasting-in-hot-Observables)
* [Unicasting in cold Observables](https://www.educative.io/courses/getting-started-with-angular/YVW5RQrDj5A#Unicasting-in-cold-Observables)

## Hot Observables

How can an Observable be hot? How can it be cold? Well, there is a distinction between when an Observable is considered hot and when it is considered cold.

A **hot Observable** is when the data that the Observable exposes is created outside the Observable.

Suppose, we have an Observable that is returning data from a backend service, maybe using an API, and this Observable emits the data as soon as it accesses it. If the call to load the data from the API is made outside of the Observable, then at the time this Observable is created, the data is already there, ready for the Observable to send. Such an Observable is considered **hot**.

### Example: click event

An example of a hot Observable is an Observable that’s created from a click event. This is considered hot because the click event data exists outside of the Observable, and the click event is created by clicking an element on the page. **If there isn’t a subscriber for this click event, the event is still fired, but nothing happens; the event is just lost**.

📝 **Note:** Press the **RUN** button to compile and serve the application. Once the app compiles, click on the URL given after Your app can be found at, and see what happens!

In the example given below, the click event is fired, but nothing happened!

* JavaScript

1

2

3

4

5

6

const myButton = document.getElementById("myButton");

/\*rxjs.fromEvent(myButton, "click").subscribe(buttonClickHandler);

function buttonClickHandler() {

  myButton.textContent = "Hello World!";

}\*/





Run

Save

Reset

## Cold Observables

A **cold Observable** is when the data it returns has been created inside of the Observable, and the Observable is being used to emit the data to its Observers.

If an Observable that uses data from an API loads this data within the Observable when it is created, then this Observable is considered cold. In a cold Observable, if it has multiple subscribers, then on every emission, **each subscriber gets its own copy of the data** being emitted, and the data is not produced until the Observable is subscribed to.

This is unlike the hot Observable, where the data is created outside of the Observable and is present even if the Observable is not subscribed to. If a hot Observable is receiving data and there are no subscribers, then the data is simply “lost”.

### Example: Observable created using from operator

An example of a cold Observable is an Observable that is created using the from operator of RxJS. This operator can create an Observable from a sequence or array of items. The cold Observable is created, and the data (the array passed into the from operator) is then emitted from the Observable when the Observable is subscribed to. Unlike the hot Observable, which uses a click event, the cold Observable does not **“lose” data** because the data is not"generated" until the Observable is subscribed to.

1

2

3

4

5

6

7

const { from } = require('rxjs');

const DATA\_SOURCE = [ '1', '2', 'hello', 'hey' ];

const observable$ = from(DATA\_SOURCE)

observable$.subscribe(console.log)





Run

Save

Reset

Observable created using from operator

So, there is a subtle difference between hot and cold Observables. It is worth knowing the difference between them, especially when we start looking at Operators in RxJS so that we know what type of Observable they create. This lets us know if the data they emit has been generated. If we know where the data an Observable emits has come from, it helps us understand our code better and reduce the risk of bugs being caused where we expect some data to be available and it’s not.

## Multicasting in hot Observables

**Multicasting** is the term given to the situation when an Observable is emitting data to multiple Observers.

Each time an Observer subscribes, the Observable starts emitting data to that Observer, and with each new Subscriber, the Observable starts an event handler and starts emitting to this new Subscriber. This means that each Subscriber gets its own version of the data being sent by the Observable. But you may not want this. Instead, you may want all the Subscribers to get the same data from the Observable and not its own version of the data. The way we can decide this is by using the hot Observable approach.

If the Observable is a hot Observable and the data is created outside of the Observable, when it is Subscribed to, the data each Subscriber gets is the same because it has come from the same source if we think of our fromEvent operator, which creates an Observable from a DOM event (a button click event or a mouse event) when the Observable emits that DOM event, all the Subscribers get the same DOM event.

## Unicasting in cold Observables

If the Observable is a cold, then each Subscriber gets its own version of the data. Cold Observables get their data when they are created and subscribed to, so on each subscription, the data the Observable emits is created. A classic example of this is if we have an Observable that creates a timer. With every Subscriber, they get their own version of the timer, starting at different times.

Hot and cold Observables, as well as multicasting, are quite advanced concepts to understand when you first start looking at Observables, but it is good to try and understand them early on before getting into RxJs and working with Observables in Angular. Because once you understand these concepts, you’ll know what Operators from RxJs to use in your application. Some Operators create hot Observables, while some create cold ones, and if you want to emit the one source of data to multiple Subscribers, you know that you need to create a hot Observable.

# What Is RxJS?

Let's take a closer look at the RxJS library and what the library provides.

**We'll cover the following**

* [The RxJs library](https://www.educative.io/courses/getting-started-with-angular/JY7Y7DXYKp9#The-RxJs-library)
* [Characteristic of reactive programming](https://www.educative.io/courses/getting-started-with-angular/JY7Y7DXYKp9#Characteristic-of-reactive-programming)
  + [Responsive to user demands](https://www.educative.io/courses/getting-started-with-angular/JY7Y7DXYKp9#Responsive-to-user-demands)
  + [Resilient to errors](https://www.educative.io/courses/getting-started-with-angular/JY7Y7DXYKp9#Resilient-to-errors)
  + [Elastic in handling the number of users](https://www.educative.io/courses/getting-started-with-angular/JY7Y7DXYKp9#Elastic-in-handling-the-number-of-users)
  + [Message-based communication between components](https://www.educative.io/courses/getting-started-with-angular/JY7Y7DXYKp9#Message-based-communication-between-components)
    - [What is a stream?](https://www.educative.io/courses/getting-started-with-angular/JY7Y7DXYKp9#What-is-a-stream?)

## The RxJs library

**RxJs** is the JavaScript implementation of [ReactiveX](http://reactivex.io/" \t "_blank), which is a library for developing asynchronous and event-driven programs using Observables. If we look at the ReactiveX website, it shows how to use **Rx** to make use of Observables in the applications we build. As part of the **Rx** library, there are various implementations of this in different languages-everything from Java to Dart. One of these implementations is **RxJs**, which is the JavaScript implementation of ReactiveX.

RxJS logo

📝 **Note:** The ReactiveX website is a great resource and is where you can find solutions to problems you may face using **RxJs**. Even though it may not be showing JavaScript examples, the concepts are still relevant for **RxJs**.

**RxJs** has been added to Angular as a way of adding support for making working with Observables easier. The **RxJs** library **provides functions and helpful JavaScript classes**, which we can use in our Angular code to create, manipulate, and manage Observables in our applications.

While **RxJs** comes included in Angular, it is **not part of the framework**. Instead, it is a separate library that can be used on its own in a none-Angular based JavaScript application or with other frameworks. If you want to use Observables within your JavaScript application, then **RxJs** is a great library that makes working with Observables easy.

Before we dive into what **RxJs** is, we need to understand more about reactive programming. By understanding this style of programming, we will have a clearer understanding of **RxJs** and the approach it takes. This will then translate to our Angular applications when we take full advantage of **RxJs** in Angular.

## Characteristic of reactive programming

When looking for a clear definition of what Reactive Programming is, there are many different explanations, from the very theoretical description in [Wikipedia](https://en.wikipedia.org/wiki/Reactive_programming) to the corporate sounding language of the [Reactive Manifesto](https://www.reactivemanifesto.org/) (yes, there is a manifesto for Reactive Programming).

### Responsive to user demands

This Reactive Manifesto states that a ‘reactive’ system needs to be responsive to a user’s demands. The program responds to action as quickly as possible.

For example, when a user clicks on a button, the application needs to respond almost immediately to the users’ click.

### Resilient to errors

It also needs to be resilient, which means if the application has an error, it has been designed so that it can handle the error and still continue working. It should be resilient to errors. The program responds even if there has been an error.

### Elastic in handling the number of users

They also state that a reactive application needs to be able to cope with high demand from users. The program is still responsive, even with a heavy workload.

If the application suddenly gets a larger number of users, all making demands from the application, it should be able to cope with this jump in user requests; it needs to be elastic in handling users’ numbers.

### Message-based communication between components

Finally, the Reactive Manifesto states that a ‘reactive’ application needs to be message based. This means that our application’s **components are loosely coupled**, and communication between components of the application is handled through the use of sending messages. The program relies on asynchronous messages to establish boundaries between components, ensuring loose coupling.

In an Angular application, this means data is passed between components using Events or Services as a message mechanism.

These “streams” are central to the reactive style of programming, so it’s worth understanding what we mean by a stream.

#### **What is a stream?**

A **stream** is a sequence of ongoing events ordered by time.

The stream can emit three things:

* A value
* An error
* A complete event

From what we’ve already discussed so far, this isn’t new to us. We have seen how Observer objects handle these three events through the next(), error(), and complete() events.

The stream will keep sending out these values until there are no more to send or an error occurs. These values are sent asynchronously, one after the other, until they have all been sent. Listening to this stream is what we call subscribing, that is, subscribing to an Observable.

# Observer Design Pattern

Let's explore the methodology behind RxJs.

**We'll cover the following**

* [Making use of the Observer Design pattern](https://www.educative.io/courses/getting-started-with-angular/7DZ7OMmYqQ8#Making-use-of-the-Observer-Design-pattern)
  + [Problems solved by an Observer design pattern](https://www.educative.io/courses/getting-started-with-angular/7DZ7OMmYqQ8#Problems-solved-by-an-Observer-design-pattern)
  + [Advantages of Observer design pattern](https://www.educative.io/courses/getting-started-with-angular/7DZ7OMmYqQ8#Advantages-of-Observer-design-pattern)
* [Angular and reactive programming](https://www.educative.io/courses/getting-started-with-angular/7DZ7OMmYqQ8#Angular-and-reactive-programming)
  + [Responsive program](https://www.educative.io/courses/getting-started-with-angular/7DZ7OMmYqQ8#Responsive-program)
  + [Handle errors](https://www.educative.io/courses/getting-started-with-angular/7DZ7OMmYqQ8#Handle-errors)
  + [Handle larger workloads](https://www.educative.io/courses/getting-started-with-angular/7DZ7OMmYqQ8#Handle-larger-workloads)
  + [Message-driven approach](https://www.educative.io/courses/getting-started-with-angular/7DZ7OMmYqQ8#Message-driven-approach)
* [Using RxJs in non-Angular applications](https://www.educative.io/courses/getting-started-with-angular/7DZ7OMmYqQ8#Using-RxJs-in-non-Angular-applications)
  + [Example](https://www.educative.io/courses/getting-started-with-angular/7DZ7OMmYqQ8#Example)

## Making use of the Observer Design pattern

RxJS uses the Observer Design pattern, an official design pattern from the **Gang of Four**.

📝 **Note:** The **Gang of Four** is in reference to a **group of four developers**, Erich Gamma, Richard Helms, Ralph Johnson, and John Vlissides, who in 1994 worked together on a book called Design Patterns: **Elements of Reusable object-orientated Programming**. Here, they set out 23 design patterns or approaches for writing good object-orientated software, which, in their opinion, had some major problems at that time.

This book has become a classic computer programming book, and now over 25 years later, the approaches they described in this book are still being used, including the Observer Design pattern.

### Problems solved by an Observer design pattern

The Observer Design pattern solves the following problems:

* A one-to-many dependency between objects should be defined without making the objects tightly-coupled.
* It should be ensured that when one object changes state, an open-ended number of dependent objects are updated automatically.
* It should be possible that one object can notify an open-ended number of other objects.

The Observer pattern aims to solve the problem where we have one object that needs to inform many other objects of any changes, whether they are data changes or errors.

Reactive programming uses this Observer design pattern by allowing us as developers to create data streams and then set any number of Observer objects to listen to this data over time. If this stream of data changes, then the Observer objects react to this.

### Advantages of Observer design pattern

Before this approach, the objects in our applications were more closely coupled objects that were more aware of each other. This could lead to problems with the scalability, speed, and maintenance of our applications. Now, through using the Observer pattern, our objects are loosely coupled, allowing us to **write faster, more maintainable code** because it is easier to refactor code that is loosely coupled.

## Angular and reactive programming

### Responsive program

Going back to Angular, we’ve seen that through our use of Observables, we can create many Observer objects listening to the one Observable, and we’ve seen that when data is loaded from an API call via an Observable, the data is available straight away, making our programs responsive.

### Handle errors

We’ve seen that with the error() handler of an Observer, if there are any problems, our applications can respond to an error, and the stream of data will continue to be sent from an Observable.

### Handle larger workloads

As our applications are not making requests every time, we want data to be sent from the stream that an Observable makes. The application can handle making more requests, making them able to handle larger workloads.

### Message-driven approach

Finally, by adhering to a more Reactive approach, our Angular applications use messages to keep the application more loosely coupled, helping with the long-term maintenance and speed of the application.

From what we’ve learned about Reactive Programming, it is clear to see that using this approach in Angular brings many benefits. We can see why the Angular team decided to use RxJs as a way to add this Reactive Programming approach to Angular.

Thankfully, the Angular team has decided that RxJs and Reactive Programming bring so many advantages to modern web applications that they have made RxJs a core part of Angular, so we don’t need to install it separately.

## Using RxJs in non-Angular applications

While we are focused on Angular in this book, it is worth knowing that RxJs is not just for Angular – it can also be used in a pure JavaScript application, such as a **Node application**.

### Example

If you have a Node app that creates your API for your Angular application, and you still want to make use of Observables and the asynchronous events in your JavaScript, you can. All you need to do is install RxJs via npm into your Node application.

# RxJS Operators

Let's explore what operators are, and the different categories of operators RxJS provides.

**We'll cover the following**

* [What are the operators?](https://www.educative.io/courses/getting-started-with-angular/RLrR61vn88R#What-are-the-operators?)
* [Categories of Operators](https://www.educative.io/courses/getting-started-with-angular/RLrR61vn88R#Categories-of-Operators)

## What are the operators?

**Operators** are functions that allow us to write more elegant and easier to read asynchronous code.

These operator functions are designed to be declarative so that when we’re using them, it is clear from the code what we’re doing. This helps us when we are working on a large codebase or as part of a team on a large project, where various developers share the code.

## Categories of Operators

In RxJs, there are many Operator functions, all of which have been grouped by category. This helps when we’re looking for an Operator to help with a piece of code we are trying to write. The categories are as follows:

| **Category** | **Purpose** |
| --- | --- |
| **Combination** | Help join data from multiple Observables |
| **Conditional** | Perform conditional tasks with Observables |
| **Creational** | Help create Observables |
| **Error Handling** | Provide ways of dealing with errors |
| **Multicasting** | Make Observables hot since they are cold in RxJs by default |
| **Filtering** | Take the response from an Observable and filter it |
| **Transformation** | Help transform or change the source of an Observable |
| **Utility** | A helpful set of Operators |

Having these categories not only helps when looking for an Operator for our code but helps when learning them too.

# Creational Operators

Let's explore RxJS creational operators with the help of interactive examples.

**We'll cover the following**

* [What are creational operators?](https://www.educative.io/courses/getting-started-with-angular/mEDWgpZn4mG#What-are-creational-operators?)
* [interval() operator](https://www.educative.io/courses/getting-started-with-angular/mEDWgpZn4mG#interval()-operator)
  + [Example](https://www.educative.io/courses/getting-started-with-angular/mEDWgpZn4mG#Example)
* [fromEvent() operator](https://www.educative.io/courses/getting-started-with-angular/mEDWgpZn4mG#fromEvent()-operator)
  + [Example](https://www.educative.io/courses/getting-started-with-angular/mEDWgpZn4mG#Example)
* [create() operator](https://www.educative.io/courses/getting-started-with-angular/mEDWgpZn4mG#create()-operator)
  + [Example](https://www.educative.io/courses/getting-started-with-angular/mEDWgpZn4mG#Example)

## What are creational operators?

Creational Operators are probably the ones we will use the most.

Creational operators allow us to create Observables from a number of sources.

The following operators are under the creational category:

Creational operators

That’s 10 different ways of creating an Observable – there are so many options that it’s hard not to use an Observable in our Angular applications.

## interval() operator

The interval() operator takes the time interval in milliseconds from its input and generates a new Observable on each of the duration periods we supply.

For example, if we pass in 1 second, then a new Observable will be created every second.

### Example

The example given below shows how this will work:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

const { interval } = require("rxjs");

// Emit value in sequence after every 1 second

// 1 second = 1000ms

const source = interval(1000);

// Output: 0,1,2,3,4,5,....

const subscribe = source.subscribe((val) => console.log(val));

// Unsubscribe from observable after 10s

// Output: 0,1,2,3,4,5,6,7,8,10 second has been passed

setTimeout(() => {

  subscribe.unsubscribe();

  console.log("10 second has been passed");

}, 10000);





Run

Save

Reset

interval( ) operator

## fromEvent() operator

We can use the fromEvent() operator to generate a new Observable from an event.

### Example

A click event shows how we can use this operator:

📝 **Note:** Press the **Run** button. You will see the button in the output. Press it and see what happens!

* JavaScript

1

2

3

4

5

6

7

8

const myButton = document.getElementById("myButton");

// Create observable that emits click events

const source = rxjs.fromEvent(myButton, 'click');

// Output (example): 'Event time: 7276.390000000001'

const subscribe = source.subscribe(val => console.log("Button has been clicked"));





Run

Save

Reset

Console

Clear

fromEvent( ) operator

Every time you press the Click Me! button, you will see a new Observable is created, and you will see Button has been clicked printed on the console!

## create() operator

We’ve seen the two previous examples, but the original create() operator is still a very good operator that’s easy to use.

### Example

All we need to do is use the Observable class of RxJS and call the create() operator on this class, like this:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

const { Observable } = require("rxjs");

/\*

Create an observable that emits 'Hello' and 'World' on

subscription.

\*/

const hello = Observable.create(function (observer) {

  observer.next("Hello");

  observer.next("World");

  observer.complete();

});

//output: 'Hello'...'World'

const subscribe = hello.subscribe((val) => console.log(val));





Run

Save

Reset

create( ) operator

This is a straightforward way of creating an Observable, and this example clearly shows how the next() and complete() event handlers are defined.

# Conditional Operators

Let's explore RxJS conditional operators with the help of interactive examples.

**We'll cover the following**

* [What are conditional operators?](https://www.educative.io/courses/getting-started-with-angular/xoN1Jpz9Azz#What-are-conditional-operators?)
* [defaultIfEmpty() operator](https://www.educative.io/courses/getting-started-with-angular/xoN1Jpz9Azz#defaultIfEmpty()-operator)
  + [Example](https://www.educative.io/courses/getting-started-with-angular/xoN1Jpz9Azz#Example)
* [iif() operator](https://www.educative.io/courses/getting-started-with-angular/xoN1Jpz9Azz#iif()-operator)
  + [Example](https://www.educative.io/courses/getting-started-with-angular/xoN1Jpz9Azz#Example)

## What are conditional operators?

**Conditional operator** helps us in making a decision if a specific condition needs to be met.

## defaultIfEmpty() operator

The first one we’re going to look at is the defaultIfEmpty() operator. This operator will emit a value if nothing is returned by a source Observable. If something is returned by the source Observable, then that value is emitted.

### Example

A classic example of this is demonstrated by the following code example:

* JavaScript

1

2

3

4

5

6

7

8

9

10

11

12

13

const myButton = document.getElementById("myButton");

const clicks = rxjs.fromEvent(myButton, "click");

// Unsubscribe from the observable after 1s

const clicksBeforeOne = clicks.pipe(

  rxjs.operators.takeUntil(rxjs.interval(1000))

);

// Display "no clicks" if button is not clicked within 1s

const result = clicksBeforeOne.pipe(rxjs.operators.defaultIfEmpty("no clicks"));

result.subscribe((x) => console.log(x));





Run

Save

Reset

Console

Clear

defaultIfEmpty( ) operator

Here we have used pipe() function, which allows us to combine multiple functions in a single function, and takeUntil() operator, which emits value from the source observable until the notifier observable emits its first value.

This example shows the defaultIfEmpty operator being used if the Observable from the fromEvent() operator is not emitted in time. So, in this scenario, no clicks value is emitted.

The defaultIfEmpty() operator is helpful for those times when we want to make sure that a value is emitted, then you can handle what the value is. An example of where this may be useful is if we are using an Observable to return some data from a backend. If there is an issue or no matching results, we could just return a default Observable, which our application uses to inform the end-user what has happened in their request.

## iif() operator

Another conditional operator is the iif() operator. This operator helps us decide which Observable to use when we subscribe.

### Example

If we have two Observables that both use a flag so that the system knows which one to use and the flag is set to true, the first Observable is used when we subscribe. If it’s set to false, the second Observable is used when we subscribe. The iif() operator takes in the conditional statement (the statement that checks the flag) and the source Observables. The result of this conditional statement determines what Observable to use as the source when subscribing. This is how it works:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

// RxJS v6+

const { iif, of } = require('rxjs');

let subscribeToFirst;

const firstOrSecond = iif(() => subscribeToFirst, of('first'), of('second'));

// Logs: "first"

subscribeToFirst = true;

firstOrSecond.subscribe(value => console.log(value));

// Logs: "second"

subscribeToFirst = false;

firstOrSecond.subscribe(value => console.log(value));





Run

Save

Reset

iff( ) operator

In this example, we set up the conditional statement with the two Observables (using the of( ) operator to create an Observable). Then, the flag is set each time before we subscribe. This shows how Observables can be set up.

# Combination Operators

Let's explore RxJS combination operators with the help of interactive examples.

**We'll cover the following**

* [merge operator](https://www.educative.io/courses/getting-started-with-angular/x1gmr9RxrkB#merge-operator)
  + [Example](https://www.educative.io/courses/getting-started-with-angular/x1gmr9RxrkB#Example)
* [concat operator](https://www.educative.io/courses/getting-started-with-angular/x1gmr9RxrkB#concat-operator)
  + [Example](https://www.educative.io/courses/getting-started-with-angular/x1gmr9RxrkB#Example)
* [concatAll() operator](https://www.educative.io/courses/getting-started-with-angular/x1gmr9RxrkB#concatAll()-operator)
  + [Example](https://www.educative.io/courses/getting-started-with-angular/x1gmr9RxrkB#Example)

The first operator we’re going to look at is the merge() operator.

## merge operator

The merge operator takes in a series of Observables to create one that can be subscribed to.

This can be useful if we have two Observables that return different data that we want to combine into one Observable. For example, if we have two Observables being returned from two different API calls, we could use the merge operator to combine that returned data into one data source.

### Example

Here’s a simple example of the merge operator:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

const { mapTo } = require("rxjs/operators");

const { interval, merge } = require("rxjs");

// Create three Observables using a timer

// Returns value after every 1s

const firstTimer = interval(1000);

// Returns value after every 2s

const secondTimer = interval(2000);

// Returns value after every 5s

const thirdTimer = interval(5000);

// Merge all three timer Observables into one data source

const example = merge(

  firstTimer.pipe(mapTo("FIRST!")),

  secondTimer.pipe(mapTo("SECOND!")),

  thirdTimer.pipe(mapTo("THIRD"))

);

// Subscribe to the merged Observable

const subscribe = example.subscribe((val) => console.log(val));

// Unsubscribe from merged observable after 21s

setTimeout(() => {

  subscribe.unsubscribe();

  console.log("21 seconds has been passed");

}, 21000);





Run

Save

Reset

merge( ) operator

Here, we are creating three Observables using the interval() operator, which returns an Observable using a time property. Then, we’re taking these three Observables and using the merge() operator to combine them into one, then subscribing to the result. As each internal Observable fires, we are using the map() operator to emit the constant value (either FIRST!, SECOND!, or THIRD) on the output Observable.

## concat operator

Another combination operator is called concat. This operator will take a number of Observables and return a single Observable.

### Example

Here’s how this works:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

const { of, concat } = require('rxjs');

//emits 1,2,3

const sourceOne = of(1, 2, 3);

//emits 4,5,6

const sourceTwo = of(4, 5, 6);

// emits 7,8,9

const sourceThree = of(7,8,9);

//used as static

const example = concat(sourceOne, sourceTwo, sourceThree);

//output: 1,2,3,4,5,6,7,8,9

const subscribe = example.subscribe(val => console.log(val));





Run

Save

Reset

concat( ) operator

This example creates three source Observables from a list of numbers using the of() operator. Then, it takes these three Observables and concats them into one Observable, which is subscribed to.

## concatAll() operator

Another example of a combination Operator is the concatAll() operator.

This operator takes a number of Observables, and as each one completes, it sends out its stream of data. The concatAll() operator takes the result and adds it to the next Observable in its list. This leads to a new Observable that’s created as all its source Observables complete, sending their stream of data one after the other.

### Example

This shows the concatAll() operator in action:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

const { of, interval } = require("rxjs");

const { map, concatAll } = require("rxjs/operators");

// Emit a value every 2 seconds

const source = interval(2000);

const example = source.pipe(

  // For demonstration, add 10 to and return as observable

  map((val) => of(val + 10)),

  // Merge values from inner observable

  concatAll()

);

//output: 'Example with Basic Observable 10', 'Example with Basic

//Observable 11'...

const subscribe = example.subscribe((val) =>

  console.log("Example with Basic Observable:", val)

);

// Unsubscribe from merge observable after 21s

setTimeout(() => {

  subscribe.unsubscribe();

  console.log("21 seconds has been passed");

}, 21000);





Run

Save

Reset

concatAll( ) operator

# Error Handling Operators

Let's explore RxJS error handling operators with the help of interactive examples.

**We'll cover the following**

* [Error Handling Operators in RxJS](https://www.educative.io/courses/getting-started-with-angular/JPQPMMV5NAP#Error-Handling-Operators-in-RxJS)
* [catchError() operator](https://www.educative.io/courses/getting-started-with-angular/JPQPMMV5NAP#catchError()-operator)
  + [Example](https://www.educative.io/courses/getting-started-with-angular/JPQPMMV5NAP#Example)
* [retry() operator](https://www.educative.io/courses/getting-started-with-angular/JPQPMMV5NAP#retry()-operator)
  + [Example](https://www.educative.io/courses/getting-started-with-angular/JPQPMMV5NAP#Example)
* [retryWhen() operator](https://www.educative.io/courses/getting-started-with-angular/JPQPMMV5NAP#retryWhen()-operator)
  + [Example](https://www.educative.io/courses/getting-started-with-angular/JPQPMMV5NAP#Example)

## Error Handling Operators in RxJS

All applications need error handling, and thankfully RxJS supplies a way of handling errors when working with Observables.

## catchError() operator

The catchError() operator allows us to either throw an error if we encounter an error when using a source Observable or switch to a new Observable if there is an error.

### Example

See the code given below.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

const { of } = require("rxjs");

const { map, catchError } = require("rxjs/operators");

of(1, 2, 3, 4, 5)

  .pipe(

    map((n) => {

      // when n is 4 the error is thrown

      if (n == 4) {

        throw "four!";

      }

      return n;

    }),

    catchError((err) => of("I", "II", "III", "IV", "V"))

  )

  .subscribe((x) => console.log(x));





Run

Save

Reset

catchError( ) operator

Looking at this example, we can see how the catchError() operator creates a new Observable (using an operator from the create category) that’s returned instead of the original Observable, which is producing an error.

This is great, but sometimes you may want to try to get data from an Observable that may be returning an error. As we know, Observables return a stream of items, and while it may throw an error on delivering the first item, the next item in the stream may be fine, and we still want to access the second item. We can try getting the item from the Observable again using the retry() operator.

## retry() operator

The retry() operator takes in a number, which is the number of times it should retry an action. This is extremely useful when working with API requests. We can use the retry() operator to try making the API request again if there is an error.

### Example

This example shows how the retry() operator is used:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

const { interval, of, throwError } = require("rxjs");

const { mergeMap, retry } = require("rxjs/operators");

//emit value every 1s

const source = interval(1000);

const example = source.pipe(

  mergeMap((val) => {

    //throw error for demonstration

    if (val > 5) {

      return throwError("Error!");

    }

    return of(val);

  }),

  //retry 2 times on error

  retry(2)

);

const subscribe = example.subscribe({

  next: (val) => console.log(val),

  error: (val) => console.log(`${val}: Retried 2 times then quit!`)

});





Run

Save

Reset

retry( ) operator

This is a nice example as it shows how we can retry running an Observable and still catch an error if the problem still exists after the retry.

## retryWhen() operator

The final error handling operator is the retryWhen() operator. We use this operator when we want it to retry – not based on a number of times as the retry() operator does - but when the state of the Observable matches the criteria that has been passed into the retryWhen() operator.

### Example

This is how it works:

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

//emit value every 1s

const source = interval(1000);

const example = source.pipe(

  map((val) => {

    if (val > 2) {

      //error will be picked up by retryWhen

      throw val;

    }

    return val;

  }),

  retryWhen((errors) =>

    errors.pipe(

      //log error message

      tap((val) => console.log(`Value ${val} was too high!`)),

      //restart in 6 seconds

      delayWhen((val) => timer(val \* 1000))

    )

  )

);

const subscribe = example.subscribe((val) => console.log(val));

//Unsubscribe from observable after 25s

setTimeout(() => {

  subscribe.unsubscribe();

  console.log("20 seconds has been passed");

}, 20000);





Run

Save

Reset

retryWhen() operator

This example shows that when there is an error – seen when the value equals 3 – then the error triggers the retryWhen() operator, which shows what the error is and adds a three-second delay val\*1000. Once this delay has run, the retryWhen() operator tries again.

The difference between this operator and the retry() operator is that it will only retry if the criteria that has been passed into it has run, while retry() just runs for a specific number of times. You have more control using retryWhen() if you have different reasons why the operator should retry calling an Observable.

# Transformation Operators

Let's explore RxJS transformation operators with the help of interactive examples.

**We'll cover the following**

* [What are the transformation operators?](https://www.educative.io/courses/getting-started-with-angular/xVZAm3L1w8J#What-are-the-transformation-operators?)
* [mergeMap() operator](https://www.educative.io/courses/getting-started-with-angular/xVZAm3L1w8J#mergeMap()-operator)
  + [Example](https://www.educative.io/courses/getting-started-with-angular/xVZAm3L1w8J#Example)
* [switchMap Operator](https://www.educative.io/courses/getting-started-with-angular/xVZAm3L1w8J#switchMap-Operator)
  + [Example](https://www.educative.io/courses/getting-started-with-angular/xVZAm3L1w8J#Example)
* [The operator decision tree](https://www.educative.io/courses/getting-started-with-angular/xVZAm3L1w8J#The-operator-decision-tree)

## What are the transformation operators?

These operators help in transforming the source of an Observable.

## mergeMap() operator

The mergeMap() operator allows us to take two source Observables and convert them into one source Observable. Why would this be useful? Well, if we have an application that retrieves data from two separate APIs, we can take the two API Observables and merge them into one single data source.

### Example

Let’s have a look at how this works:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

const { of } = require("rxjs");

const { mergeMap } = require("rxjs/operators");

// First source Observable

// of is a creational operator that creates Observable from a sequence of data.

const bookTitle = of("Getting Started With Angular");

// Merge two source Observables

const completeTitleObservable = bookTitle.pipe(

  mergeMap((value) => `${value} 10!`)

);

// Output: Create Getting Started With Angular 10!

const subscribe = completeTitleObservable.subscribe((value) =>

  console.log(value)

);





Run

Save

Reset

mergeMap( ) operator

Here, we’re creating an Observable that returns the Getting Started With Angular title, and then we’re using the pipe() operator to add this bookTitle Observable to a new one that’s been created from the mergeMap() operator. The mergeMap() operator returns an Observable, which is combined/merged with the first one. When we subscribe to this newly merged Observable, we can see the complete title.

## switchMap Operator

The switchMap operator is very similar to the mergeMap() operator. The difference is that in the mergeMap, the first Observable continues to keep getting data as part of this stream of data that we’ve spoken about. With the switchMap() Operator, the first Observable gets data and then switches to the next Observable. When it switches to the second Observable, the first one is unsubscribed, and we are no longer concerned with the stream of data from the first Observable. We have what we want from it, so now we switch to the new observable.

### Example

📝 **Note:** Press the **Run** button. You will see the button in the output. Press it, and see what happens!

* JavaScript

1

2

3

4

5

6

7

8

const myButton = document.getElementById("myButton");

const source = rxjs.fromEvent(myButton, 'click')

// Now we switch to an Observable from an  interval

const result = source.pipe(rxjs.operators.switchMap(() => rxjs.interval(1000)))

result.subscribe(x => console.log(x));





Run

Save

Reset

Console

Clear

switchMap( ) operator

On clicking the Click Me! button, you should see the values generated from 0 onwards, i.e., 0,1,2,.......

In the above example, we have an Observable that has returned from a click event on a button. Once that has run, it switches over to the interval()-based Observable, which is writing to the console every second. In this example, we are switching between the two Observables instead of merging Observables as we did in the mergeMap() example.

The switchMap() operator is useful when we want to get data from one Observable. Then, once we have the first data item that’s been returned from the Observable, we can switch to another Observable to get the data from the second Observable.

We can use these types of operators to create complex pipelines of data from various Observables (if you think about it, these Observables can return data from APIs) so that we can combine data or switch to different sources of data in one single pipeline. This is a powerful feature of Operators.

## The operator decision tree

The problem with so many operators is knowing which one to use when writing our code. Thankfully, there is a way to find the perfect Operator.

On the RxJs website, there is a section that helps you find the operator that you may want to use: the [Operator Decision Tree](https://rxjs-dev.firebaseapp.com/operator-decision-tree). It looks like this:

Operator Decision Tree

As you can see, it asks you a set of questions, and, based on the answers you select, it drills down until it can suggest an Operator you could use. This tool is extremely helpful and well worth bookmarking.

# RxJS Subjects

Let's explore Subjects, their types, and how we can use these in different scenarios.

**We'll cover the following**

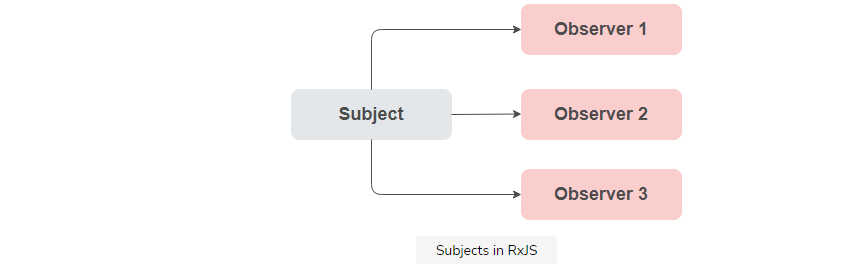
* [What is a Subject?](https://www.educative.io/courses/getting-started-with-angular/RLX10j5gAoq#What-is-a-Subject?)
* [The BehaviorSubject Observable](https://www.educative.io/courses/getting-started-with-angular/RLX10j5gAoq#The-BehaviorSubject-Observable)
  + [Example](https://www.educative.io/courses/getting-started-with-angular/RLX10j5gAoq#Example)
* [The ReplaySubject Observable](https://www.educative.io/courses/getting-started-with-angular/RLX10j5gAoq#The-ReplaySubject-Observable)
  + [Example](https://www.educative.io/courses/getting-started-with-angular/RLX10j5gAoq#Example)
* [The AsyncSubject Observable](https://www.educative.io/courses/getting-started-with-angular/RLX10j5gAoq#The-AsyncSubject-Observable)
  + [Example](https://www.educative.io/courses/getting-started-with-angular/RLX10j5gAoq#Example)

## What is a Subject?

A **Subject Observable** is a special type of Observable which is multicast, instead of unicast like the Observables we have been looking at.

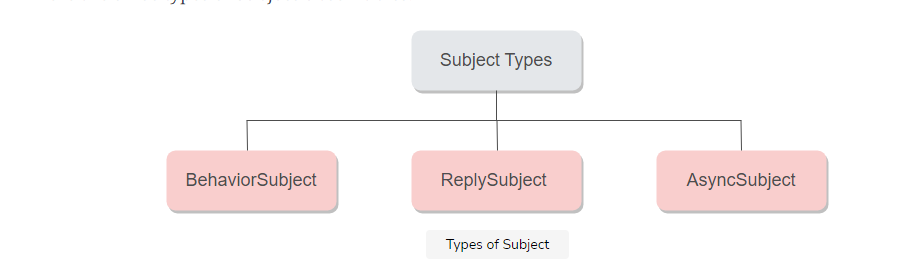
When an Observable creates a connection to an Observer, it wires up the event handlers (next, complete, and error), and then sends data to that Observer. Each Observer that is connected to the Observable is getting its own set of data. Multicasting is where one Observable sends the same data to multiple Observers. It is casting out its stream of data to all the Observers that are subscribed, and they all get the same data.

Subjects in RxJS



A good way to think of a Subject Observable is as an EventEmitter (which we discussed in the [Components, Templates, and Forms chapter](https://www.educative.io/collection/page/10370001/4603693004488704/5860980926971904)). This is the way of sending out data or an event to anyone listening. These events can be listened for by multiple EventListeners, and this is exactly what a Subject does.

There are three types of Subject Observables:



Types of Subject

Let’s look at each one.

## The BehaviorSubject Observable

BehaviorSubject stores the latest emitted value and sends that value to every new Observer that subscribes to it.

### Example

This shows how this type of Observable works:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

const { BehaviorSubject } = require('rxjs');

const subject = new BehaviorSubject(123);

//two new subscribers will get initial value => output: 123, 123

subject.subscribe(console.log);

subject.subscribe(console.log);

//two subscribers will get new value => output: 456, 456

subject.next(456);

//new subscriber will get latest value (456) => output: 456

subject.subscribe(console.log);

//all three subscribers will get new value => output: 789, 789, 789

subject.next(789);

// output: 123, 123, 456, 456, 456, 789, 789, 789





Run

Save

Reset

The BehaviorSubject Observable

The first Subscriber gets this initial data, which is set when the BehaviorSubject is created. Then on each next(), any new Subscribers get the new value. This type of Observable is very useful if you want to guarantee that when an Observer first subscribes, some data is passed.

For example, if you want to pass data to a form, and you want the form to have some initial data, the BehaviorSubject can be used.

## The ReplaySubject Observable

It sends some previous data to the newly subscribed Observers. To do this, the ReplaySubject must store a certain amount of emitted values and constantly update this data as new values are sent to the Subject.

The ReplaySubject Observable emits old values to the new Subscribers. If, for example, we have an application that shows financial data, we might use a ReplaySubject Observable to send data to a panel in a web application that shows financial changes over time. By using the ReplaySubject Observable, any old values will be sent to this panel, as well as any new values that need to be displayed.

### Example

This shows how a ReplaySubject Observable can be used:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

const { ReplaySubject } = require('rxjs');

const sub = new ReplaySubject(3);

sub.next(1);

sub.next(2);

sub.subscribe(console.log); // OUTPUT => 1,2

sub.next(3); // OUTPUT => 3

sub.next(4); // OUTPUT => 4

sub.subscribe(console.log);

// OUTPUT => 2,3,4 (log of last 3 values from new subscriber)

sub.next(5); // OUTPUT => 5,5 (log from both subscribers)





Run

Save

Reset

The ReplaySubject Observable

The value that’s passed into the ReplaySubject Observable is a buffer for how much historical data it should keep which in this example is 3.

## The AsyncSubject Observable

The last of these Subject Observables is the AsyncSubject Observable, which only sends out the last items of data that it has when the complete() handler is called. So, if we have an Observable that is returning a list of names, and the last name on this list before the complete() handler is Rogers, then the value that all Subscribed Observers get would be Rogers, even if they called the subscribe() event before the complete() handler of the AsyncSubject Observable was called.

### Example

This shows how this works:

1

2

3

4

5

6

7

8

9

const { AsyncSubject } = require('rxjs');

const sub = new AsyncSubject();

sub.subscribe(console.log);

sub.next(123); //nothing logged

sub.subscribe(console.log);

sub.next(456); //nothing logged

sub.complete(); //456, 456 logged by both subscribers





Run

Save

Reset

The AsyncSubject Observable

Since the value 456 was the last value set in the next() handler before the complete() handler was called, both the subscribe() event handlers get this last value. This is helpful if you want to guarantee that all of the Subscribed Observers get the last value when a connection to an AsyncSubject Observable is complete.

If we have an application that shows a number in the header of the application and in a panel of the application, then we want both places to show the same value when an Observable is loading a count into this number. In order to make sure both places show the same number, we may use an AsyncSubject Observable so that when the complete() handler is called, both places show the latest count.

These three types of Subject Observables all have slight differences when it comes to what value they may return to their Subscribed Observables, but it is important to remember that Subject Observables differ from standard Observables in that they send out the same data to multiple Observers. This can be useful for web applications that need to show the same data in multiple places.

# How Angular Uses RxJS

Let's explore how Observables, Operators, Subjects, and RxJs are used in our Angular applications.

**We'll cover the following**

* [Where to consider using RxJs](https://www.educative.io/courses/getting-started-with-angular/qAKYDWLEvQr#Where-to-consider-using-RxJs)
  + [Services](https://www.educative.io/courses/getting-started-with-angular/qAKYDWLEvQr#Services)
    - [Handle errors](https://www.educative.io/courses/getting-started-with-angular/qAKYDWLEvQr#Handle-errors)
    - [Format data](https://www.educative.io/courses/getting-started-with-angular/qAKYDWLEvQr#Format-data)
    - [Cancel HTTP request](https://www.educative.io/courses/getting-started-with-angular/qAKYDWLEvQr#Cancel-HTTP-request)
  + [Reactive forms](https://www.educative.io/courses/getting-started-with-angular/qAKYDWLEvQr#Reactive-forms)
  + [Components](https://www.educative.io/courses/getting-started-with-angular/qAKYDWLEvQr#Components)
    - [Handle click events](https://www.educative.io/courses/getting-started-with-angular/qAKYDWLEvQr#Handle-click-events)
    - [Create time-based Observable](https://www.educative.io/courses/getting-started-with-angular/qAKYDWLEvQr#Create-time-based-Observable)
    - [Filter list of records](https://www.educative.io/courses/getting-started-with-angular/qAKYDWLEvQr#Filter-list-of-records)

So, how does RxJs relate to Angular? Well, the Angular team has made RxJs a core part of the framework. We haven’t had to install RxJs separately as a dependency in our projects. Having RxJs as part of Angular means that we can use Observables, Observers, and Operators throughout our applications, wherever we choose to use them.

## Where to consider using RxJs

When writing our Angular applications, there are certain times when an RxJs-based approach may be a better solution than a non-RxJs-based approach.

### Services

In regards to Services and HTTP requests, as we saw in the [Dependency Injection, Services, and HttpClient chapter](https://www.educative.io/collection/page/10370001/4603693004488704/5073909983477760), RxJs is used for managing them. All the REST requests made by the HttpClient service return Observables, which we need to subscribe to at the Component level.

#### Handle errors

You can use Operators like retry() or catchError() with HTTP requests to manage when there is an issue with the request.

#### Format data

You can use the pipe() and map() operators to take the results of several HTTP requests and format the data into a structure you need for your component. For example, suppose you have a data grid that you need to populate. With the pipe() and map() Operators, you can build out a pipeline where the data from an HTTP request goes through until it is in a structure that works with your data grid. There are over 25 different Transformation category operators we can make use of when working with the data we get through our Services.

#### Cancel HTTP request

We can also cancel HTTP requests simply by calling the unsubscribe() handler of an Observable. This control over HTTP requests is not possible without Observables. Requests can’t be canceled in Promises, so knowing how to use RxJS within our Services has many benefits.

### Reactive forms

We can use RxJS in Reactive forms, which we discussed in the [Components, Templates, and Forms chapter](https://www.educative.io/collection/page/10370001/4603693004488704/6519949706133504).

By doing this, we can **observe the values** that are being entered into our forms, checking for errors or the wrong type of information (for example, email address format) as the user completes a form.

We can **store the form’s state** as the user fills in the form by observing the form fields and storing the value. This could be helpful if the user leaves the page with a form while filling it out. We are keeping track of the values entered into the form so that users don’t have to start again. They can carry on from where they left off. This helps make the experience of using your application better for the end-user.

### Components

RxJs is not just for Services and Forms. We can also make use of RxJs at the component level.

#### Handle click events

In the TypeScript classes of our components, RxJs can be used for handling click events, such as using the fromEvent() operator to create an Observable, which watches when a button is clicked.

#### Create time-based Observable

We can create time-based Observables using the interval() and timer() Operators if we need our components and templates to change state after a time period. For example, we may have a testing application where a **user takes a time-based exam**. If they haven’t completed a question in time, then the component/template warns the user. This can be done with a time-based Observable.

#### Filter list of records

We can also use the filter() operator to filter down a list of records that are displayed in a component as the user types in a search box. Without having to make an API call to get filtered data, we can filter the data we have as soon as the user starts searching.

# What is AsyncPipe?

Let's learn how we can use Async Pipe to manage the subscribing and unsubscribing of Observables.

**We'll cover the following**

* [AsyncPipe](https://www.educative.io/courses/getting-started-with-angular/7nx8KQ9wz2B#AsyncPipe)
  + [Example](https://www.educative.io/courses/getting-started-with-angular/7nx8KQ9wz2B#Example)
* [When to use AsyncPipe and when to Subscribe](https://www.educative.io/courses/getting-started-with-angular/7nx8KQ9wz2B#When-to-use-AsyncPipe-and-when-to-Subscribe)
  + [Use Subscribe if we need data in more than one place](https://www.educative.io/courses/getting-started-with-angular/7nx8KQ9wz2B#Use-Subscribe-if-we-need-data-in-more-than-one-place)
  + [Use AsyncPipe if we need data in one place](https://www.educative.io/courses/getting-started-with-angular/7nx8KQ9wz2B#Use-AsyncPipe-if-we-need-data-in-one-place)

## AsyncPipe

One important pipe that we can use in our templates is the [AsyncPipe](https://angular.io/api/common/AsyncPipe" \t "_blank). This works with Observables and helps manage the Subscribing and Unsubscribing from Observables that are used in our components.

The AsyncPipe will return the last value emitted from an Observable and tells the Observable that the value can be checked for the next value in the stream. It will also automatically unsubscribe from the Observable when the component is destroyed or no longer in the UI. This automatic unsubscribing helps **reduce the risk of memory leaks in our applications**. They can still happen if we don’t unsubscribe in our Services when using Observables, but the AsyncPipe is extremely useful when used in components.

### Example

Here is an example showing how AsyncPipe can be used:

📝 **Note:** Press the **RUN** button to compile and serve the application. Once the app compiles, click on the URL given after Your app can be found at, and see what happens!

###### /

e2e

src

app

app-routing.module.ts

app.component.scss

app.component.spec.ts

app.component.ts

app.module.ts

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.scss

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**app.component.ts**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

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

import { Observable } from 'rxjs';

@Component({

    selector: 'app-root',

    template: `<div>

                <code>observable|async</code>:

                Time: {{ time | async }}

              </div>`,

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

})

export class AppComponent {

  time: Observable<string>;

  constructor() {

    this.time= new Observable(observer => {

      setInterval(() => {

        observer.next(new Date().toString());

      }, 1000);

    });

  }

}





Run

Save

Reset

**Your app can be found at:**[https://ed-6434592358924288.educative.run](https://ed-6434592358924288.educative.run/)

The inline template

In the preceding code, there is no Subscribe or Unsubscribe event calls. This is handled by the AsyncPipe, which is always set just after the place in the template where the value of the Observable is displayed.

## When to use AsyncPipe and when to Subscribe

It’s easy to think that, with AsyncPipe, we can have all our Observables managed. We could use AsyncPipe to subscribe and unsubscribe within the template of our component.

We could, but there are scenarios where we wouldn’t want to do this.

For example, if we have multiple source Observables that we need to map together using Operators. We may need to have these Observables return data within the ngOnInit() life cycle hook of a component in order to have data returned before the component has completed loading. In this scenario, we have to subscribe within ngOnInit() . We can’t use AsyncPipe because it’s too late in the component life cycle.

### Use Subscribe if we need data in more than one place

So, if we need the data from an Observable in more places than just the template of a Component, the Subscribe method is a better choice. The problem with using Subscribe is that we need to handle unsubscribing to the Observable within our code.

### Use AsyncPipe if we need data in one place

If we have a collection of data that we want to loop over with ngFor, the AsyncPipe is ideal for managing the subscribing and unsubscribing to it.

<ul>  
    <li \*ngFor="let person of people$ | async">  
        {{person.firstname}}  
    </li>   
</ul>

Here, we are looping over a collection of people, and this collection is returned from an Observable. AsyncPipe is used to subscribe to this people$ collection, and once the list of names has been written to the template, the AsyncPipe unsubscribes.

The benefit of the Async pipe is that we can go through Observables’ results directly within templates. The problem is if we then want to use the same Observable data as a non-template part of a component, we would need to set up a new Observable that we can subscribe to in our code.

# Creating Reactive Forms for Client Contacts Manager Application

Let's use RxJS to improve the search of our application and add onChanges lifecycle hook to populate a Reactive form with the data passed in from the parent Component.

**We'll cover the following**

* [Major changes](https://www.educative.io/courses/getting-started-with-angular/qZ0X1jZ6Zy3#Major-changes)
* [The client folder](https://www.educative.io/courses/getting-started-with-angular/qZ0X1jZ6Zy3#The-client-folder)
  + [Amend client.service.ts](https://www.educative.io/courses/getting-started-with-angular/qZ0X1jZ6Zy3#Amend-client.service.ts)
  + [Update client-detail](https://www.educative.io/courses/getting-started-with-angular/qZ0X1jZ6Zy3#Update-client-detail)
* [The company folder](https://www.educative.io/courses/getting-started-with-angular/qZ0X1jZ6Zy3#The-company-folder)
  + [Amend company.service.ts](https://www.educative.io/courses/getting-started-with-angular/qZ0X1jZ6Zy3#Amend-company.service.ts)
  + [Update company-detail](https://www.educative.io/courses/getting-started-with-angular/qZ0X1jZ6Zy3#Update-company-detail)

## Major changes

In this version of an application, we have made the following changes:

* Updated both the Client and Company services to show that Observables are being returned from the HttpClient calls.
* Updated both the Client Edit form and the Company Edit form to Reactive forms.
* Used RxJs in the Search Form Component, so it now starts searching as you type.
* Added OnChanges lifecycle hook to both the Client and Company edit forms in order to populate a Reactive form with the data passed in from the parent Component.

## The client folder

We will make the following changes in the client folder.

### Amend client.service.ts

Amending this file to now return the Observable from each method allows the Observable to be Subscribed to when used. This starts to show how Observables are used in different ways, trying to introduce Subscribing.

### Update client-detail

We have expanded the form so that it is populated with data when the component loads. It checks for a change to the Client details using the onChange event. If there are new Client’s details to display, then the form is populated with these new details.

📝 **Note:** Similarly, we have made some minor changes in client-edit-page, client-details-page, client-page, and client-form, search-form components, and shared.module. These minor changes are highlighted within their respective components.

## The company folder

We will make the following changes in the company folder.

### Amend company.service.ts

The return type for all the methods in this service are set as Observables, so they can be subscribed to.

### Update company-detail

Updated this so when a new Company’s details are loaded, the form, which is a child component of the company-detail component, is populated. Also, a new Company Object is created when a new Company’s details are added via the form. This object is then saved via the Service layer to the application.

📝 **Note:** Similarly, we have made some minor changes in the company-search-page.component, and company-details-page, company-page components. These minor changes are highlighted within their respective components.

###### /

e2e

src

app

about

clients

client-detail

client-detail.component.html

client-detail.component.scss

client-detail.component.spec.ts

client-detail.component.ts

client-details-page

client-edit-page

client-form

client-list

client-page

client-search-page

client.module.ts

client.service.spec.ts

client.service.ts

client.ts

company

navigation

page-not-found

search

services

shared

app-routing.module.ts

app.component.html

app.component.scss

app.component.spec.ts

app.component.ts

app.module.ts

custom-material.module.ts

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.scss

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**client-detail.component.html**

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

    <button (click)="delete(client)" mat-raised-button color="warn">Delete</button>

  </mat-card-footer>

</mat-card>

<mat-card \*ngIf="editView">

  <form [formGroup]="clientEditForm">

  <mat-card-header>

    <mat-form-field>

      <input matInput type="text" name="firstname" formControlName="firstname" />

    </mat-form-field>

    <mat-form-field>

      <input matInput type="text" name="lastname" formControlName="lastname" />

    </mat-form-field>

  </mat-card-header>

  <mat-card-subtitle>

    <h2>{{ client?.companyName }}</h2>

  </mat-card-subtitle>

  <mat-card-content>

    <mat-form-field>

      <input matInput type="text" name="email" formControlName="email" />

    </mat-form-field>

    <mat-form-field>

      <input matInput type="text" name="telephone" formControlName="telephoneNumber" />

    </mat-form-field>

  </mat-card-content>

  <mat-card-footer>

    <button mat-button mat-raised-button color="primary" (click)="save()">Save</button>

  </mat-card-footer>

</form>





Run

Save

Reset

**Your app can be found at:**[https://ed-6434592358924288.educative.run](https://ed-6434592358924288.educative.run/)

Client Contacts Manager Application with RxJS

📝 **Note:** Press the **RUN** button to compile and serve the application. Once the app compiles, you can see its output in the **Output** tab or by clicking on the URL given after Your app can be found at.

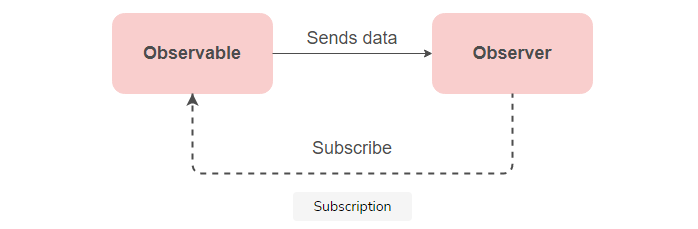
# Summary: Observables and RxJS

Here is a quick summary for you!

We have covered a lot in this chapter. It has been one of the largest in this course, but it is worth understanding all that we have covered in this chapter in order to be able to write fast and powerful Angular applications.

The use of **Observables and RxJs** is an extremely important part of how we build applications with Angular. With the need for modern web applications to be extremely fast and responsive to end-user demands, being able to design and build applications that make use of Observables means that we can create Angular applications that react to the user’s needs.

Subscription



With all that we have learned in this chapter, we can now take our knowledge and start looking at how we can handle state management within our Angular applications and how the new library, NgRx, makes use of Observables and Observers to manage state within applications.

# State Management and NgRx

Let's learn why state management is needed in Angular applications.

**We'll cover the following**

* [Problems of complex Angular applications](https://www.educative.io/courses/getting-started-with-angular/gxQvEzY1lGD#Problems-of-complex-Angular-applications)
* [How can we manage the state?](https://www.educative.io/courses/getting-started-with-angular/gxQvEzY1lGD#How-can-we-manage-the-state?)
* [Why choose NgRx?](https://www.educative.io/courses/getting-started-with-angular/gxQvEzY1lGD#Why-choose-NgRx?)

In this chapter, we’re going to be exploring two aspects of Angular development. While very important, they are not parts of everyday Angular development, but they are worth looking into if we want to build larger-scale applications with Angular. These two aspects of enterprise-level Angular application development are **state management** and **NgRx**.

## Problems of complex Angular applications

As we build more and more complex business applications using Angular, we eventually run into issues, such as:

* how to manage all the interaction between components
* how to persist data between sections of the application
* how to make one part of our application access data from another
* how changing from one part of the app affects another.

All of these problems come under the responsibility of state management.

## How can we manage the state?

Luckily for us, as Angular developers, there are a few options regarding how we can manage state. We could roll our own solution **using RxJs**, or we could make use of **local storage** and a **service layer** to store data (but this isn’t a good approach).

Thankfully, some very smart people in the Angular community have come up with approaches to solving this problem. There are many frameworks available for us that help manage state in Angular, such as **NgXs**, **Akita**, and **NgRx**.

## Why choose NgRx?

The reason we’re looking at NgRx in this course, instead of other state management libraries, is that NgRx is used in some large enterprise-level applications. Therefore, by learning NgRx, you’ll be able to take what you’ve learned from this course and put that knowledge to work on larger applications. This will lead your career as an Angular developer, and you will be able to work on these large enterprise-level applications that Angular specializes in.

After you have read this chapter and learned about what state management is, it may be worth looking at the other solutions.

In this chapter, we will learn about the following topics:

* The problems we face and why state management is the answer
* Why state management is important in modern web applications
* What NgRx is and why you should consider using it
* How NgRx helps implement a solution for adding state management to an Angular application
* What the Redux pattern is
* How to add NgRx to an application
* How to use the features of NgRx to manage the state of an Angular application

State management and NgRx are both large topics that could each fill a course on their own, so we will just go through the basics and become familiar with both.

📝 **Note:** Press the **RUN** button to compile and serve the application. Once the app compiles, you can see its output in the **Output** tab or by clicking on the URL given after Your app can be found at.

###### /

e2e

src

app

models

store

app-routing.module.ts

app.component.html

app.component.scss

app.component.spec.ts

app.component.ts

app.module.ts

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.scss

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**app.component.ts**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

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

import { select, Store, State } from '@ngrx/store';

import { Observable } from 'rxjs';

import { ActionTypes } from './store/actions/manage-apples.actions';

import { State as ApplesState } from './store/reducers';

@Component({

  selector: 'app-root',

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

  styleUrls: ['./app.component.scss']

})

export class AppComponent {

  title = 'NgRx Fruit Shop';

  applesInStore$: Observable<any> = this.store.select(state => state.apples);

  constructor(private store: Store<ApplesState>) {}

  addApples(apples: number): void {

    this.store.dispatch({

      type: ActionTypes.SaveApples,

      payload: apples

    });

  }

  removeApples(apples: number): void {

    this.store.dispatch({

      type: ActionTypes.RemoveApple,

      payload: apples

    });

  }





Run

Save

Reset

**Your app can be found at:**[https://ed-6434592358924288.educative.run](https://ed-6434592358924288.educative.run/)

Angular application with Selectors

See how the **“Total Number of Apples in Store”** changes by adding and removing **Apples**.

# Defining State Management

Learn what state management is, aspects of the modern web application that comes under state management, and the types of states.

**We'll cover the following**

* [What is state management?](https://www.educative.io/courses/getting-started-with-angular/NEJVwxPnKRz#What-is-state-management?)
* [Modern web application and state management](https://www.educative.io/courses/getting-started-with-angular/NEJVwxPnKRz#Modern-web-application-and-state-management)
* [Types of states](https://www.educative.io/courses/getting-started-with-angular/NEJVwxPnKRz#Types-of-states)

## What is state management?

**State management** is the approach we take to manage how data is synchronized, stored, and accessed throughout our application—making sure that the data displayed to the user is current and correct.

## Modern web application and state management

A lot of aspects of a modern web application can come under the term state management. With modern web applications, we need to think about how we manage data from multiple sources, such as:

* How do we **pass data** around our application? Whether we use **Events** to pass data or **Services**.
* What data should we pass **within the URL**, or should we even pass data in a URL?
* How do we make sure that all the data we show in the application is in sync? For example, if we’re building a financial application. What if a user sees a monetary value on one screen and the same represented value is a different number because the data is not in sync, then the application appears broken.

Data is synced

* We also need to make sure that the state of the application represents what the state is on a backend server. If our application loads data from a backend service and the backend gathers data from multiple sources, which it then sends out to our application, how do we make sure that our application reacts when this updated data is sent so that it shows the same data as the backend application?

Application reacts to data changes

## Types of states

There are a few types of states that we need to be aware of, as follows:

* The **Local User Interface state**.
* **URL-based state**, where values are persisted through query parameters attached to URLs in the application.
* **Server-side state**, which is state information that’s persisted on a backend/database and sent to the application.
* **Client-side state**, where the state information is stored on the client (the web browser) instead of a backend server.
* **Transient state**, where the state is stored on the client-side, but the user is not aware of it. They do not see values being passed in the URLs.
* **Persisted state**, where the state stored in a backend server is passed and stored on the client by using services and local data storage.

As you can see, there are a lot of different types of states that we have to manage within applications. It has become far more of a challenge than just managing data between UI components. Without thinking about how we will handle state management within our application, trying to manage all these various states can lead to more and more complex code, which could be difficult to read and test.

# Methods for Implementing State Management

Learn what state management is and how we can implement state management in our Angular applications.

**We'll cover the following**

* [Approaches to state management](https://www.educative.io/courses/getting-started-with-angular/3w3AY1jpw7Q#Approaches-to-state-management)
  + [Using @Input and @Output decorators](https://www.educative.io/courses/getting-started-with-angular/3w3AY1jpw7Q#Using-@Input-and-@Output-decorators)
  + [Services and Dependency Injection](https://www.educative.io/courses/getting-started-with-angular/3w3AY1jpw7Q#Services-and-Dependency-Injection)
  + [Pass state information through route parameters](https://www.educative.io/courses/getting-started-with-angular/3w3AY1jpw7Q#Pass-state-information-through-route-parameters)
  + [Store data in the browser’s local storage](https://www.educative.io/courses/getting-started-with-angular/3w3AY1jpw7Q#Store-data-in-the-browser%E2%80%99s-local-storage)
  + [Observable-based approach](https://www.educative.io/courses/getting-started-with-angular/3w3AY1jpw7Q#Observable-based-approach)
  + [Redux pattern and RxJs](https://www.educative.io/courses/getting-started-with-angular/3w3AY1jpw7Q#Redux-pattern-and-RxJs)
  + [NgRx](https://www.educative.io/courses/getting-started-with-angular/3w3AY1jpw7Q#NgRx)

## Approaches to state management

There are a few ways we can tackle the issue of how to implement state management within our Angular applications. Some examples of how we could handle state management within an Angular application are given below:

* Manage the passing of state information through **@Input and @Output decorators** in our Components, along with Events to send state information between components in the application.
* Make use of **Services** and **Dependency Injection** to pass state information between Components and other Services.
* We could pass state information through URLs via **Route Parameters**.
* We can pass data around by storing data in the browser’s **local storage**.
* Use an **Observable-based approach**. For example, create Services where the state is stored and accessed through Observables that react to changes in the application.
* Using an RxJs approach based on the Redux pattern (which we will look at soon).
* Use a third-party library like NgRx, which has been designed to solve this problem.

### Using @Input and @Output decorators

The first approach is a perfectly reasonable approach. The state is passed into and out of components using the @Input and @Output decorators, which we discussed in the [Components, Templates, and Forms chapter](https://www.educative.io/collection/page/10370001/4603693004488704/6556515983949824). When this state data is passed into the Component, it can change the view in the Template, and if the user makes a change to this state data within that Template, this can be transmitted to other areas of the application through Events and the EventEmitter.

This approach is fine, but as the logic of the application becomes more and more complex, this approach can soon become a maintenance nightmare and lead to **spaghetti code**.

Input and Output decorators

### Services and Dependency Injection

Using Services and Dependency Injection is a better approach since the state is stored and accessed through the Services, which, as we know from the [Dependency Injection, Services, and HttpClient chapter](https://www.educative.io/collection/page/10370001/4603693004488704/6233226233249792), means that any Component or other Service can access this state data when a Service has been injected.

Dependency Injection

However, again, this could lead to a very complex code base, and you would have to manage the state being passed through Promises. However, making use of Observables would make the state more reactive to changes.

### Pass state information through route parameters

We can also make use of what the browser provides us with. We could pass state information through URLs via Route Parameters, which we learned about in the [Routing and Navigation chapter](https://www.educative.io/collection/page/10370001/4603693004488704/5920025016795136). Route parameters allow us to attach data to the end of a URL. This data can be accessed as route parameters in different sections of the application.

Route

While this approach is fine for a small amount of data, as the data grows in size, parameters can become complex, thus making it difficult to keep track of the data being passed.

Another problem with this approach is that this data can be seen in the browser’s address bar, which could be a security risk, depending on the data being passed.

### Store data in the browser’s local storage

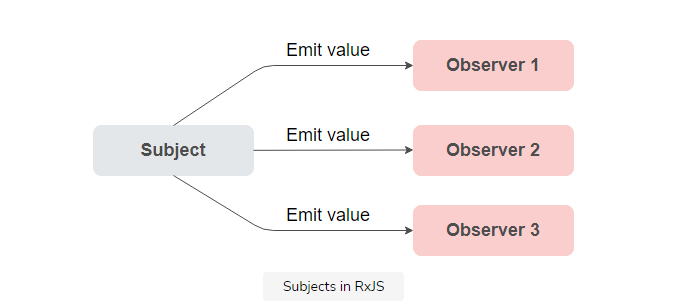
One other approach we could use to pass data around the application is by storing data in the browser’s local storage. All modern browsers have **local** and **session storage**, which we can write to using JavaScript. So, if we wanted to store some sort of state, we could write it to either local storage or session storage. Both are great options. There are a few TypeScript libraries we can add to our Angular applications that make writing to these storage options really easy.

Browser local storage

The issue with these two approaches is that, again, managing this can become more and more complex.

### Observable-based approach

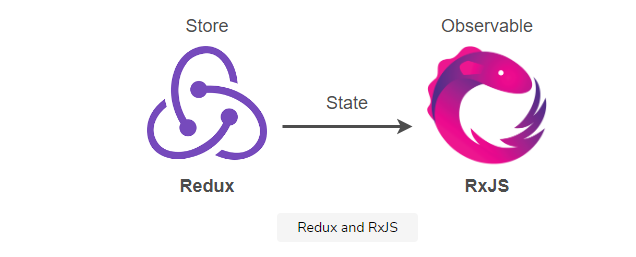
An Observable-based approach is a good approach because we can Subscribe to Observables, which then emit state changes to all the Subscribed Observers. Using Subject Observables such as ReplaySubject and BehaviorSubject provides the ability to send out state-based changes to multiple Observers, which is a better approach.



As an application grows in complexity, coming up with ways to solve state-management problems is difficult. Using a proven pattern helps speed up development because the approach we take to manage state as we build parts of an application have been clearly defined. A developer who follows a standard approach for dealing with state management can implement new features of an application faster than if they need to roll their own approach.

### Redux pattern and RxJs

Using the Redux pattern and RxJs is a good approach because we are making use of **Observables** to broadcast state changes within our application, and following a design pattern like **Redux** gives us a roadmap to follow when implementing state management in our applications.

This approach’s drawback is that a less experienced developer may implement this approach differently compared to an experienced developer who has learned about the best approaches for implementing state management following the Redux pattern.

### NgRx

This is why using a third-party library like NgRx is a great way forward. Not only are we following the principles set out in the Redux pattern, but the approach set out in NgRx has been written by experienced Angular developers who have solved this complex issue many times and add solutions to NgRx based on their knowledge.

# All of these approaches have pros and cons, and there are alternatives to using NgRx, but it is a very common approach in the Angular ecosystem and worth looking into. The Redux Library

Let's explore the Redux library and its three principles.

**We'll cover the following**

* [What is Redux?](https://www.educative.io/courses/getting-started-with-angular/x1OYBMyr8jE#What-is-Redux?)
* [The principles of Redux](https://www.educative.io/courses/getting-started-with-angular/x1OYBMyr8jE#The-principles-of-Redux)
  + [Single source of truth](https://www.educative.io/courses/getting-started-with-angular/x1OYBMyr8jE#Single-source-of-truth)
  + [Example: Finance application](https://www.educative.io/courses/getting-started-with-angular/x1OYBMyr8jE#Example:-Finance-application)
  + [The state is read-only](https://www.educative.io/courses/getting-started-with-angular/x1OYBMyr8jE#The-state-is-read-only)
  + [Changes are made through pure functions](https://www.educative.io/courses/getting-started-with-angular/x1OYBMyr8jE#Changes-are-made-through-pure-functions)
    - [What is a pure function?](https://www.educative.io/courses/getting-started-with-angular/x1OYBMyr8jE#What-is-a-pure-function?)

## What is Redux?

**Redux** is a state container for JavaScript apps.

It is closely linked to the React framework, but it can and is being used by other JavaScript frameworks, including Angular. If we go to the [ReduxJS website](https://redux.js.org/" \t "_blank), we’ll see that there are a set of principles that Redux has.

These principles have also been implemented in NgRx. We will look at how NgRx has implemented these principles later. First, let’s discuss these principles.

## The principles of Redux

There are three principles of Redux:

Principles of Redux

Now, let’s explore each of these in turn to get a better understanding of what each means.

### Single source of truth

In the Redux approach, the state of the application is known as the **single source of truth**.

This means that the state of the application is stored in one place so that whenever we get some state information for our application, we only get it from a single place.

This solves the problem where we may have two components that are showing the state information. They are both getting it from the same place, which means that both components will show the same data.

There is no way one component can show the same type of state information as another component, as the data in that state information is different in both places. Sharing the data from the one store solves this problem.

### **Example: Finance application**

Think of a finance application. If two parts of an application are showing the same type of financial data, the values are wrong because they are loading this data from different sources. This could be a bug in the application, and the end-user could lose confidence in the application. Having a single source of truth means this wouldn’t happen.

### The state is read-only

The second principle is that the state is read-only. This means that the state store can only be read directly. It can’t be changed directly. All changes must be made through our **Reducers**, which use pure functions to amend the state.

Using this Reducer approach allows all changes to be centralized and happen one after the other in order. This means there is never an issue with the data stored in the state being changed by one part of an application before another part of the application can update the state. We never get what is called a **race condition**, where one change goes through before a previous change has happened.

The state is changed through Reducer

### Changes are made through pure functions

The last of the three principles is that changes are made through pure functions.

#### **What is a pure function?**

A **pure function** is a function that takes in an argument and always returns a value.

These are very common in the Math() library of JavaScript. For example, the Floor() function is a pure function. It takes in a number and returns a number that is the floor of that number. Within a pure function, there are no changes too or side effects of using the pure function. You pass in a value and get a value. There are no local properties of the function that affect the returned value.

Principles of Redux

In Redux, we have Reducers (again, another concept we will look at in more depth when exploring NgRx), which are pure functions that take the state and an Action, and return the next state. So, if we have an Action that adds an object to the state, the Reducer would take the object in the Action and return the current state with the new object as a new state. We pass in the current state, and the Action object and the Reducer would return a single return value, which is the state and Action object reduced down to a single state object.

How Reducer changes the state

These concepts are high level, and as we explore NgRx through code examples, you’ll see how these have been put into practice. What we need to remember is that the Redux library approach to state management is to have one single place where all the state information is kept within an application. This single place can only be amended through one approach and not directly from the view or elsewhere in the application. Changes to the state can only be made using a consistent approach.

Keeping the state separate and only accessible through a clearly defined approach means that the state of our applications cannot be amended or overwritten through some other means. This guarantees that the data in our state is current and correct.

# Exploring NgRx

Let's explore NgRx and its features in detail to understand its core concepts.

**We'll cover the following**

* [What is NgRx?](https://www.educative.io/courses/getting-started-with-angular/JQvQLoxBKMg#What-is-NgRx?)
* [The core concepts of NgRx](https://www.educative.io/courses/getting-started-with-angular/JQvQLoxBKMg#The-core-concepts-of-NgRx)
  + [Store](https://www.educative.io/courses/getting-started-with-angular/JQvQLoxBKMg#Store)
  + [Reducers](https://www.educative.io/courses/getting-started-with-angular/JQvQLoxBKMg#Reducers)
  + [Actions](https://www.educative.io/courses/getting-started-with-angular/JQvQLoxBKMg#Actions)
  + [Selectors](https://www.educative.io/courses/getting-started-with-angular/JQvQLoxBKMg#Selectors)
* [Using NgRx concepts in Angular](https://www.educative.io/courses/getting-started-with-angular/JQvQLoxBKMg#Using-NgRx-concepts-in-Angular)
  + [Example: Save form data](https://www.educative.io/courses/getting-started-with-angular/JQvQLoxBKMg#Example:-Save-form-data)

## What is NgRx?

Now that we know what Redux is and what the principles of Redux are, we can start taking a detailed look at NgRx and go through the features of NgRx in order to understand it.

The best place to start when looking into NgRx is on the [official website](https://ngrx.io/).

NgRx looks different from the official Angular website. This is because while NgRx is used with Angular, it isn’t officially part of Angular. It’s a library that we can use with Angular.

📝 **Note:** If you are interested in setting up NgRx locally, you can go through the following lesson: [Installing NgRx](https://www.educative.io/collection/page/10370001/4603693004488704/5611949462126592)

NgRx

## The core concepts of NgRx

There are four key concepts in NgRx, and they are as follows:

Key concepts in NgRx

### Store

The store is the **state container** of our application, which is an Observable object. This object contains all the actual state information for our application. The Actions and Reducers interact with this store to add this state management to the application.

### Reducers

Reducers are the **pure functions** we mentioned earlier when looking at [Redux](https://www.educative.io/collection/page/10370001/4603693004488704/4943528466579456). Reducers handle the functionality where we go from one state to the next. A Reducer function will take in the current state plus an Action and then return a new state.

For example, if we have an Action that adds a new item to the state, a Reducer would return a new state that contains this new item as one single state object.

### Actions

Actions are the events we dispatch from our **Components** and **Services**. These Action events or Actions are unique events that happen in our application. They cover everything from a user clicking a button to a Service making an API call. All these different things are triggered by Actions.

### Selectors

Selectors are also pure functions that return a selection of the State store object. They allow us to get information for the State store without having to return the entire store. These are helpful when getting state information for our views, where we don’t want to have the entire State Store just to show a small amount of data in the view level.

How NgRx manages the state

## Using NgRx concepts in Angular

In an NgRx based Angular application, we write Actions, Reducers, and Selectors in order to access the Store object that contains all the state information of our application.

### Example: Save form data

Suppose we want to write some data to the store when the user clicks on a button, for example, a **Save button**. We write an Action that contains a payload of the form data. This gets passed to a Reducer, which looks at the action and performs this action on the state, which in this case will be an ADD action.

📝 **Note:** A payload is an object that is passed to the Reducers, along with the action from the action event. It can contain any data that is needed as part of the action.

In the upcoming lesson, let’s have a look at some examples of each of these different concepts so that we know what they look like.

# Example of an Action

Let's explore what Action is with the help of an example.

**We'll cover the following**

* [What is Action?](https://www.educative.io/courses/getting-started-with-angular/7DoyPE0B0pA#What-is-Action?)
  + [Action interface](https://www.educative.io/courses/getting-started-with-angular/7DoyPE0B0pA#Action-interface)
  + [Action example](https://www.educative.io/courses/getting-started-with-angular/7DoyPE0B0pA#Action-example)
  + [app.component.html](https://www.educative.io/courses/getting-started-with-angular/7DoyPE0B0pA#app.component.html)
  + [app.component.ts](https://www.educative.io/courses/getting-started-with-angular/7DoyPE0B0pA#app.component.ts)

## What is Action?

An **Action** is dispatched when a message needs to be sent.

### Action interface

All Actions are based on the Action interface, which defines what an Action must have. The interface is like the signature of the class. It defines what the class should have without setting the implementation. A class that implements an interface has to implement what the interface says it should have, but how the class implements that feature is up to the individual class.

The Action interface of NgRx looks like this:

interfaceAction{  
  type: string;   
}

Any class that implements the Action interface must have a type. That’s all it really needs to be an Action.

### Action example

This is an example of an Action:

{  
  type: 'Add Apples'  
  payload: {  
    numberOfApples: number  
  }  
}

This Action has a type called “Add Apples”, which is the Action it is performing. In the payload object, we’re passing the number of apples, which is to be saved. This can be written in a separate TypeScript file called models/apples.ts.

We can create separate TypeScript files for each Action if we want and have them kept in an actions folder. Or, we could also have all our actions grouped by functionality in a single TypeScript file. So, if we had further actions that are related to managing apples, we could have a TypeScript file called manage-apples.actions.ts. Here, we could keep all the Apple basket-related actions:

📝 **Note:** Press the **RUN** button to compile and serve the application. Once the app compiles, click on the URL given after Your app can be found at, and see what happens!

When the application compiles, you can open the given link to view your application. Then right-click on the application page, click inspect, move to the console tab, and see how clicking on each button dispatches the respective Action.

###### /

e2e

src

app

models

store

actions

manage-apples.actions.ts

app-routing.module.ts

app.component.html

app.component.scss

app.component.spec.ts

app.component.ts

app.module.ts

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.scss

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**manage-apples.actions.ts**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

import { Action } from '@ngrx/store';

import { Apples } from '../../models/apples';

export enum ActionTypes {

  SaveApples = 'Save Apples',

  RemoveApple = 'Remove Apple',

  RemoveAllApples = 'Remove All Apples'

}

export class SaveApples implements Action {

  readonly type = ActionTypes.SaveApples;

  constructor(public payload: Apples) {}

}

export class RemoveApples implements Action {

  readonly type = ActionTypes.RemoveApple;

  constructor(public payload: Apples) {}

}

export class RemoveAllApples implements Action {

  readonly type = ActionTypes.RemoveAllApples;

  constructor(public payload: Apples) {}

}

// This is used in the Reducer as a list of the Actions it can support

export type ActionUnion = SaveApples | RemoveApples | RemoveAllApples;





Run

Save

Reset

**Your app can be found at:**[https://ed-6434592358924288.educative.run](https://ed-6434592358924288.educative.run/)

Angular application with Action

In this TypeScript class, we have all the Actions associated with managing the apples in an application. At the top, we’re using an enum to define the Action type. This helps reduce spelling errors when setting the type of Action and produces cleaner looking code.

To dispatch an Action, we have made the following changes in our application:

### app.component.html

In this file, we have added three buttons. On a button click, an event is triggered, and it will dispatch the respective Action.

<button (click)="addApples(1)">Add an Apple</button>  
  <button (click)="removeApples(1)">Remove an Apple</button>  
  <button (click)="removeAllApples()">Remove All Apples</button>

### app.component.ts

To dispatch this Action we use the Store’s dispatch method, like this:

this.store.dispatch({  
  type: ActionTypes.SaveApples,  
  payload: apples  
});

# Example of a Reducer

Let’s explore what a Reducer is in NgRx.

**We'll cover the following**

* [What is a Reducer?](https://www.educative.io/courses/getting-started-with-angular/q2LJ6n92q0R#What-is-a-Reducer?)
  + [Example](https://www.educative.io/courses/getting-started-with-angular/q2LJ6n92q0R#Example)
* [Reducing file size for Actions and Reducers](https://www.educative.io/courses/getting-started-with-angular/q2LJ6n92q0R#Reducing-file-size-for-Actions-and-Reducers)

## What is a Reducer?

**Reducers** handle the functionality where we go from one State to the next.

A Reducer doesn’t implement an interface as the Action does, but it does return a State object. The Reducer function takes in two arguments, that is, the State and the Action.

### Example

For our Managing Apples example, a Reducer could look as follows:

First, we define the interface of our State model in apples-reducer.ts:

export interface State {  
     applesCount: number;  
}

Then, we create an initial state object, along with some default values:

export const initialState : State = {  
    applesCount: 1;  
}

Then, we create a Reducer function that handles different Actions:

###### /

e2e

src

app

models

store

state

reducers

apples-reducer.ts

actions

app-routing.module.ts

app.component.html

app.component.scss

app.component.spec.ts

app.component.ts

app.module.ts

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.scss

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**apples-reducer.ts**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

import { ActionTypes, ActionUnion } from '../actions/manage-apples.actions';

import { Action } from '@ngrx/store';

export interface State {

  applesCount: number;

}

export const initialState: State = {

  applesCount: 1

};

export function reducer(state = initialState, action: Action): State {

  switch (action.type) {

    case ActionTypes.SaveApples: {

      console.log(state);

      return {

        ...state,

        applesCount: state.applesCount + 1

      };

    }

    case ActionTypes.RemoveApple: {

      console.log(state);

      return {

        ...state,

        applesCount: state.applesCount - 1

      };

    }

    case ActionTypes.RemoveAllApples: {

      console.log(state);

      return {





Run

Save

Reset

**Your app can be found at:**[https://ed-6434592358924288.educative.run](https://ed-6434592358924288.educative.run/)

Angular application with Reducer

In this Reducer function, you can see that we’re using a switch statement, which uses the ActionUnion list from our Actions file to match against the Action being fired. Then, when the case statement matches the Action type, it returns the State with the updated message. If the Action does not have a matching case statement, the State is returned without any Actions performed on it.

This is a very simple example of a Reducer, but the important thing to note is the use of the enums when defining Action names. This really helps because our modern IDEs and editors can use this information to tell us what Actions are available as we type. Therefore, we don’t have to keep jumping back and forth between Action files and Reducer files, trying to see what things are named.

## Reducing file size for Actions and Reducers

Note the use of the case/switch statement to handle the different Actions within the Reducer and how we create a different case for each Action. The more Actions we have, the more case Statements we have. You would think this could lead to large TypeScript files with hundreds of case statements. But by grouping the Actions into separate files – for example, manage-apples.action.ts or manage-user.action.ts – we only have the Actions for a specific part of our application within the one file. With the Reducer file having case statements for all the Actions within the single Actions file, we have a lot of the application state management within just two separate files: one for Actions and one for Reducers.

# Example of the Store

Let's explore what a Store is with the help of an example.

**We'll cover the following**

* [What is a Store?](https://www.educative.io/courses/getting-started-with-angular/RLz2Q4n6K60#What-is-a-Store?)
* [Creating a State tree of Reducers](https://www.educative.io/courses/getting-started-with-angular/RLz2Q4n6K60#Creating-a-State-tree-of-Reducers)
  + [Example](https://www.educative.io/courses/getting-started-with-angular/RLz2Q4n6K60#Example)

## What is a Store?

So far, we’ve mentioned State and Store, but sometimes it’s difficult to know what the difference between the two is. Let’s try to clear up the distinction between the two.

A **Store** is an Object from NgRx that contains the State of the application. This Store object has an API that allows us to manage the State contained within the Store.

The **State** is the data we create for our application, which is specific to the application we’re building. So, from our apple example earlier, applesCount is the state information, which we keep in a Store object.

To create this State, we first create an interface of the State, defining the properties of the State model. There is an example of this in the ApplesState interface that we defined earlier. This interface says that the ApplesState has a property of applesCount, so when we create the initial state object, we have to add it (TypeScript will throw an error if we don’t).

Finally, we tell the Reducer to return a state object of Type ApplesState. As you can see, the State is created within the associated Reducer (See line 40 of apples-reducer.ts).

###### /

e2e

src

app

models

store

state

apples.state.ts

reducers

actions

app-routing.module.ts

app.component.html

app.component.scss

app.component.spec.ts

app.component.ts

app.module.ts

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.scss

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**apples.state.ts**

1

2

3

4

5

6

7

8

9

10

import { Apples } from 'src/app/models/apples';

export interface ApplesState  {

  applesCount: number;

}

export const initialState: ApplesState = {

  applesCount: 0

};





Run

Save

Reset

**Your app can be found at:**[https://ed-6434592358924288.educative.run](https://ed-6434592358924288.educative.run/)

Angular application with State

## Creating a State tree of Reducers

In an NgRx-based application, we probably have many Reducer files that all contain state information, but we want a way to combine all the Reducers into one state tree. A state tree is a mapping of all the **State objects contained** within our **Reducer files**. This idea of a tree is similar to how we think of Components within Angular and how they all branch out of a module.

### Example

In order to create one of these State trees, we need to create a new file in our Reducers folder called index.ts and export all the State through this one index file. For example, our ApplesState could be added to an index.ts as:

###### /

e2e

src

app

models

store

state

reducers

index.ts

apples-reducer.ts

actions

app-routing.module.ts

app.component.html

app.component.scss

app.component.spec.ts

app.component.ts

app.module.ts

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.scss

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**index.ts**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

import \* as appleReducer from './apples-reducer';

import { ApplesState } from '../state/apples.state';

import { ActionReducerMap, MetaReducer } from '@ngrx/store';

export interface State {

  apples: appleReducer.State;

}

export const reducers: ActionReducerMap<State> = {

  apples: appleReducer.reducer

};

export const metaReducers: MetaReducer<State>[] = [];





Run

Save

Reset

**Your app can be found at:**[https://ed-6434592358924288.educative.run](https://ed-6434592358924288.educative.run/)

Angular application with the State tree

There is a couple of things going on here. First, we are creating an interface of our State, which contains ApplesState. If we had another Reducer file that manages the State of oranges, it would be added to State like this:

export interface State {  
   apples: applesProductState.ApplesState,  
   oranges: orangesProductState.OrangeState  
}

This interface can grow to have all the State information. Next, we pass this interface to an exported constant variable called a Reducer, which is using a utility from NgRx called ActionReducersMap. This mapper helps us build up a list of the Reducers (and their State). But in order to make use of type checking, we pass the State interface as the type of the Reducer’s Object being exported. This means that we can’t add new Reducers that are not part of the State interface. So, let’s try adding a banana state to the reducers map, like this:

export const reducers: ActionReducerMap<State> = {  
    apples: applesProductState.reducer,  
    oranges: orangesProductState.reducer,  
      
    // not part of theState interface  
    banana: bananaProductState.reducer  
}

We’ve created interfaces defining our State within our Reducers and then created an initial state object in the Reducer. Then, we’ve created a new index.ts where we have created an interface defining what the State objects of the section of our application are structured on. Then we created a state tree object that maps all the Reducers we’ve created for this main State section.

If you run the application given above, you will notice that our application is not aware of this State object. So we need to make an application aware of State at the module level.

# Registering the State

Let's explore how we can register the State at global or feature level.

**We'll cover the following**

* [Methods for registering the State](https://www.educative.io/courses/getting-started-with-angular/qVpD765Zk5G#Methods-for-registering-the-State)
  + [Registering the State at the global level](https://www.educative.io/courses/getting-started-with-angular/qVpD765Zk5G#Registering-the-State-at-the-global-level)
    - [Example](https://www.educative.io/courses/getting-started-with-angular/qVpD765Zk5G#Example)
  + [Registering the State at the feature level](https://www.educative.io/courses/getting-started-with-angular/qVpD765Zk5G#Registering-the-State-at-the-feature-level)
    - [Example](https://www.educative.io/courses/getting-started-with-angular/qVpD765Zk5G#Example)
* [Recap of the steps involved in defining the State in NgRx](https://www.educative.io/courses/getting-started-with-angular/qVpD765Zk5G#Recap-of-the-steps-involved-in-defining-the-State-in-NgRx)

## Methods for registering the State

There are two levels where we can register the State:

* at the global application level
* at the feature level.

### Registering the State at the global level

The global level means that the entire application can access the state tree made from the Reducers. This is fine for a small application, but if we have a more complex application, we could use the feature level module to register the state tree for that feature.

#### Example

To register at the global level, we use the StoreModule class from the NgRx Store API to make NgRx aware of the Reducers file we’ve just created. This would look like this:

📝 **Note:** Press the **RUN** button to compile and serve the application. Once the app compiles, click on the URL given after Your app can be found at, and see what happens!

When the application compiles, you can open the given link to view your application. Then right-click on the application page, click inspect, move to the console tab, and see how the State is changed with the button click.

###### /

e2e

src

app

models

store

app-routing.module.ts

app.component.html

app.component.scss

app.component.spec.ts

app.component.ts

app.module.ts

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.scss

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**app.module.ts**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

import { BrowserModule } from '@angular/platform-browser';

import { NgModule } from '@angular/core';

import { AppRoutingModule } from './app-routing.module';

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

import { StoreModule } from '@ngrx/store';

import { reducers, metaReducers } from '../app/store/reducers';

@NgModule({

  declarations: [

    AppComponent

  ],

  imports: [

    BrowserModule,

    AppRoutingModule,

    StoreModule.forRoot(reducers, { metaReducers })

  ],

  providers: [],

  bootstrap: [AppComponent]

})

export class AppModule { }





Run

Save

Reset

**Your app can be found at:**[https://ed-6434592358924288.educative.run](https://ed-6434592358924288.educative.run/)

Angular application with State registered at the global level

This is fine, but a better option is to use the NgRx Store forFeature() function instead, which allows us to register the State at the feature module level, instead of just at the global module level. This way, we can use lazy-loading to load State when a feature of our application is being used, instead of having to load it all when the application starts up.

### Registering the State at the feature level

As we discussed in [NgModules chapter](https://www.educative.io/collection/page/10370001/4603693004488704/4518520179130368" \t "_blank), we can divide an application into modules – one per feature. So, if our application allows users to order fruit in one section and order drinks in another, we would have a feature module for the order fruit section and another for the order drinks section. These are called **feature modules**.

#### Example

Again, using our Apple example, we need to create a state at the feature module level. In app.module.ts, we set up the state like this:

import { NgModule } from '@angular/core';  
import { StoreModule } from '@ngrx/store';  
  
@NgModule({  
    imports: [StoreModule.forRoot({})],  
})  
export class AppModule {}

Here, we’re using NgRx’s StoreModule and creating an empty object, while in the previous example we loaded in the appleReducer.

Now, in the appleOrdering feature module, we use NgRx’s StoreModule forFeature() method to add the State, like this:

import { NgModule } from '@angular/core';  
import { StoreModule } from '@ngrx/store';  
import { appleReducer } from './reducers/fruit.reducer';  
     
@NgModule({  
    imports: [  
        StoreModule.forFeature('appleOrdering', appleReducer)  
    ],  
})  
export class AppleOrderingModule {}

Here, we add the state that was set out in our fruit reducers. Next, we need to make the global state aware of this, so back in the app.module.ts file, we add the following code:

import { NgModule } from '@angular/core';  
import { StoreModule } from '@ngrx/store';  
import { FruitOrderingModule } from './ordering/fruit-ordering.module';  
  
@NgModule({  
    imports: [  
        StoreModule.forRoot({}),   
        FruitOrderingModule  
    ],  
})  
export class AppModule {}

Here, we have registered our feature module with the main global module and used the StoreModule of NgRx to make NgRx aware of the State held within our FruitOrderingModule.

## Recap of the steps involved in defining the State in NgRx

So far, we’ve gone through how we define the state in our applications and make the NgRx Store aware of it, as well as the Actions we can perform on this State. Let’s recap what steps are involved in setting this up:

1. Create an Actions file defining the Events we can perform on the State (manage-apples.actions.ts).
2. Create a Reducers file containing an interface that defines the structure of the State (apples-reducer.ts).
3. Add an initial state object to the Reducer file of the State.
4. Add a switch statement to the Reducer file, with a case for each Action that can be performed on the State.
5. Always return the State from the Reducer file.
6. Create an interface defining the structure of the map of Reducers for a feature.
7. Create an ActionReducerMap of all the Reducers in a feature of the application.
8. Create a global level Store using the NgRx StoreModule's forRoot() method in app.module.ts.
9. Register a feature level Store using the NgRx StoreModule's forFeature() method.
10. Register the feature level Module Store with the global level Store.

This is a lot of steps that need to be taken in order to create a State and define how we interact with it. But if we think about having a large, complex application that has a lot of state information, having a consistent, step by step approach that NgRx takes leads to code that is clean and consistent for developers to read and understand how the application works.

# Example of Selectors

Let's explore how we can use Selector to access the information from the state store.

**We'll cover the following**

* [What is the Selector?](https://www.educative.io/courses/getting-started-with-angular/xV775mBogK9#What-is-the-Selector?)
* [NgRx helper functions](https://www.educative.io/courses/getting-started-with-angular/xV775mBogK9#NgRx-helper-functions)
  + [createFeatureSelector() method](https://www.educative.io/courses/getting-started-with-angular/xV775mBogK9#createFeatureSelector()-method)
  + [createSelector() method](https://www.educative.io/courses/getting-started-with-angular/xV775mBogK9#createSelector()-method)
* [Getting State using a Selector](https://www.educative.io/courses/getting-started-with-angular/xV775mBogK9#Getting-State-using-a-Selector)
  + [Step 1: Create a Selector](https://www.educative.io/courses/getting-started-with-angular/xV775mBogK9#Step-1:-Create-a-Selector)
  + [Step 2: Subscribe to Results](https://www.educative.io/courses/getting-started-with-angular/xV775mBogK9#Step-2:-Subscribe-to-Results)

## What is the Selector?

The **Selector** allows us to get information for the State store without having to return the entire store.

Selectors are pure functions in NgRx, similar to Reducers.

## NgRx helper functions

NgRx provides two helper functions: createSelector() and createFeatureSelector()

### createFeatureSelector() method

The createFeatureSelector() method gets the feature state (the state of a feature module),

### createSelector() method

The createSelector() method returns a selection of that State information. As you can see, we can use the first method to get the State of a feature and the second type of Selector to get a slice of data from that first State.

Both of these methods are from the NgRx Selector API. What’s really clever about these two helper functions is that when NgRx sees we’re running a Selector with the same two arguments we used previously to get state data, instead of going off and repeating what NgRx has just done, it just returns the same State as it did before without the round trip to go and get the data. The NgRx Store keeps track of the Selectors that have been used, and if we use one that it knows of, it invokes that Selector again without having to run the same selector code again. This gives us a great **performance boost**.

## Getting State using a Selector

Let’s look at an example of creating a Selector to retrieve data from our ApplesState.

### Step 1: Create a Selector

Here, we are creating a selector using createSelector() method that returns a selection of the state information.

import { createSelector } from '@ngrx/store';  
  
import { ApplesState } from '../state/apples.state';  
  
const selectApples = (state: ApplesState) => state;  
  
export const selectApplesFromStore = createSelector(  
  selectApples,  
  (state: ApplesState) => state.applesCount  
);

### Step 2: Subscribe to Results

To use these Selectors in a component, we import these Selectors into our component and then subscribe to the results using the Async pipe in the app.component.html:

📝 **Note:** Press the **RUN** button to compile and serve the application. Once the app compiles, click on the URL given after Your app can be found at, and see what happens!

###### /

e2e

src

app

models

store

app-routing.module.ts

app.component.html

app.component.scss

app.component.spec.ts

app.component.ts

app.module.ts

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.scss

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**app.component.ts**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

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

import { select, Store, State } from '@ngrx/store';

import { Observable } from 'rxjs';

import { ActionTypes } from './store/actions/manage-apples.actions';

import { State as ApplesState } from './store/reducers';

@Component({

  selector: 'app-root',

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

  styleUrls: ['./app.component.scss']

})

export class AppComponent {

  title = 'NgRx Fruit Shop';

  applesInStore$: Observable<any> = this.store.select(state => state.apples);

  constructor(private store: Store<ApplesState>) {}

  addApples(apples: number): void {

    this.store.dispatch({

      type: ActionTypes.SaveApples,

      payload: apples

    });

  }

  removeApples(apples: number): void {

    this.store.dispatch({

      type: ActionTypes.RemoveApple,

      payload: apples

    });

  }





Run

Save

Reset

**Your app can be found at:**[https://ed-6434592358924288.educative.run](https://ed-6434592358924288.educative.run/)

Angular application with Selectors

See how the **“Total Number of Apples in Store”** changes by adding and removing **Apples**.

In this example, we make use of the Store class from NgRx so that we can use the select() method in order to invoke the Selector. Then, we subscribe to the Observable returned from the select() method. You will notice that we’re not using a Service to get the data as we may have done if we weren’t using NgRx. This is because we are now using the Selectors to get data from the State instead of using a Service to load in data.

This does raise the question of, if we aren’t using Services to load data into our smart components, how do we handle loading data from external sources such as APIs? This is where Effects come into play, which we will look at next.

# Effects

Let's explore how we can load data from external APIs using Effects in NgRx.

**We'll cover the following**

* [What is the Effect?](https://www.educative.io/courses/getting-started-with-angular/Bn8lDz09GlQ#What-is-the-Effect?)
* [Key concepts of Effects](https://www.educative.io/courses/getting-started-with-angular/Bn8lDz09GlQ#Key-concepts-of-Effects)
* [Writing an Effect](https://www.educative.io/courses/getting-started-with-angular/Bn8lDz09GlQ#Writing-an-Effect)
  + [Dissection of Effect service](https://www.educative.io/courses/getting-started-with-angular/Bn8lDz09GlQ#Dissection-of--Effect-service)
* [Registering an Effect](https://www.educative.io/courses/getting-started-with-angular/Bn8lDz09GlQ#Registering-an-Effect)
  + [Register Effect at the module level](https://www.educative.io/courses/getting-started-with-angular/Bn8lDz09GlQ#Register-Effect-at-the-module-level)
  + [Register Effect at the feature level](https://www.educative.io/courses/getting-started-with-angular/Bn8lDz09GlQ#Register-Effect-at-the-feature-level)

## What is the Effect?

**Effects are RxJs-based**, so bringing in our knowledge of RxJs from the [Observables and RxJs chapter](https://www.educative.io/collection/page/10370001/4603693004488704/4959879964917760):

The role of an **Effect** is to listen to any Action that has been dispatched.

After doing this, the Effect checks to see if it has a case for the dispatched Action, in the same way as a Reducer. If there is a case for the Action, the Effect will run that case. This could be making an API call to either get or send data.

Then, the result of the API call would cause the Effect to emit another Action, which a Reducer would pick up, taking us back into the NgRx workflow of Actions and Reducers. So, Effects make calls to the side of the main NgRx workflow, which is why they get the name Side Effects or Effects.

Role of Effects in NgRx

## Key concepts of Effects

There are some key concepts of Effects, which are as follows:

* They isolate side effects from the components
* They are **long-running Services** that listen to an Observable of every Action dispatched from the Store
* They filter these Actions by the **Type** of the Action using an RxJs Operator
* They perform both **synchronous** and **asynchronous tasks** and then return an Action from that task, which is picked up by a Reducer

The main thing to pick up from Effects is that they perform tasks outside of the Store, Action, and Reducer workflow of NgRx in order to run either synchronous or asynchronous tasks, then return from this outside task and dispatch a new Action, which brings us back into the Store, Action, and Reducer NgRx workflow.

## Writing an Effect

An Effect is a Service class that is injected using Dependency Injection. When writing an Effect, it’s good practice to keep them in an effects folder and use the common naming convention of subject.effect.ts, for example, products.effects.ts or users.effect.ts.

### Dissection of Effect service

An Effect service is made up of the following parts:

* The createEffect() function
* An injectable Actions service
* The list of Actions are filtered using the ofType() operator
* Effects are subscribed to the Store Observable that NgRx provides
* Services that access external APIs are injected into the Effect service

Let’s have a look at an example Effect service so that we can see these parts in action:

import { Injectable } from '@angular/core';  
import { Actions, Effect, typeOf } from  '@ngrx/effects';  
import { \* } as FruitActions from './fruit.actions.ts';  
  
@Injectable()  
export class ProductsEffects {  
    constructor(private actions$: Actions,   
        private productService:ProductService) {}  
  
    @Effect()  
    loadProducts$ = createEffect( this.actions$.pipe(  
       ofType(FruitAction.LoadFruit).map(() =>  
           this.productService.getAllProducts().pipe( map (   
           products => ({  
               type: 'Products Loaded', payload: products  
            }))  
        }  
        ...

In this simple Effect’s Service, there are a number of things going on. First, we’re using the @Injectable operator to state that this class is a Service, and we’re also using @Effect(), an operator from NgRx, which tells NgRx that this is an Effect Service. In the constructor, we’re passing in the Actions service and the Product Service (our Service for returning data).

Within the Effect operator section, we’re using the injected action$ service to find the invoked Action, and then perform an action and dispatch a new Action with the results from the Service.

The injected Actions service contains a stream of the possible Actions of the application. We’re using the ofType() operator to check what the name of the current Action being dispatched from the Store is. If there is a match, the code for that Action is run, which in this case is a call to the ProductService. When the Service function has run, the results are mapped (using the RxJs map() operator), and a new Action is dispatched back to the Store so that NgRx is aware of this new Action. The new Action has both a type and a payload. This payload contains the results of the Service call, which in this example is a list of products.

Once we’ve created an Effect service, we need to register it so that NgRx is aware that we register this Effect in the same way we register Selectors.

## Registering an Effect

Like Selectors, there are two places where we can register an Effect: either in the main app.module.ts file or at the feature level.

### Register Effect at the module level

Taking the ProductsEffect that we created earlier, when registering it at the main app.module.ts level, it will work like this:

import { EffectsModule } from '@ngrx/effects';  
import { ProductsEffect } from './effects/products.effects.ts';  
  
@NgModule({  
   imports: [  
       EffectsModule.forRoot([ProductsEffect])  
   ],  
})  
  
export class AppModule {}  
...

Here, we’re importing the EffectsModule from NgRx, and in our main @NgModule, we use the EffectsModule.forRoot() method to register our newly created ProductsEffect service. You can see that the forRoot() method accepts an array of Effect Services, so we could pass in a list of these Effect Services and register them.

### Register Effect at the feature level

We can also just register Effects at the feature level. This brings the benefit of Angular not having to load all the Effect services when the application starts. Through lazy loading, when a feature is being used, the Effects for that feature are loaded at the same time, reducing the need to load everything at a startup.

To register an Effect at the feature level, we use the forFeature() method of the EffectsModule of NgRx in the same way we register Selectors at the feature level. So, in a ProductsModule, we would register the same products.effects.ts file like this:

import { EffectsModule } from '@ngrx/effects';  
import {ProductsEffect } from './effects/products.effects.ts';  
     
@NgModule({  
   imports: [  
       EffectsModule.forFeature([ProductsEffect])  
   ]  
})  
  
export class ProductsModule {}

# Testing Angular Applications

Let's learn about Testing Angular Applications in this chapter.

**We'll cover the following**

* [Chapter learning outcomes](https://www.educative.io/courses/getting-started-with-angular/JE7MW6L802D#Chapter-learning-outcomes)

## Chapter learning outcomes

In this chapter, we are going to look at **testing**, which is an essential part of developing an enterprise-level application. We will briefly discuss the importance of writing tests and the benefits of following a **Test-Driven Development (TDD)** approach. We will then look at how Angular supports testing by reviewing the test spec files that Angular generates automatically through the CLI. Then, we’ll see how to run our tests to check if they have all passed or failed.

Testing Angular application

After reading this chapter, you will have learned the following:

* What tests are and what Testing and Test Driven Development (TDD) is.
* Why tests are important.
* What Jasmine and Karma are.
* How to write tests using Jasmine.
* How to run tests and see the output of tests using Karma.
* How to check if your tests either pass or fail.
* How Angular creates and runs tests.
* What features Angular provides to make testing easier.

There is a lot to know about testing in both Angular and in general web development, so we best get going!

# Testing and Test Driven Development

Let's explore why testing and test-driven development are important and why we use Jasmine for writing tests in Angular.

**We'll cover the following**

* [Why is Test-driven development important?](https://www.educative.io/courses/getting-started-with-angular/qVlw10zDG3R#Why-is-Test-driven-development-important?)
* [Jasmine: Behavior-Driven JavaScript](https://www.educative.io/courses/getting-started-with-angular/qVlw10zDG3R#Jasmine:-Behavior-Driven-JavaScript)
* [Angular and Jasmine](https://www.educative.io/courses/getting-started-with-angular/qVlw10zDG3R#Angular-and-Jasmine)

## Why is Test-driven development important?

In the world of Angular, testing is extremely important. As we know from when we explored the **Dependency Injection (DI)** in [Dependency Injection, Services, and HttpClient chapter](https://www.educative.io/collection/page/10370001/4603693004488704/6233226233249792), data is passed into our Components using DI. This leads to our components being isolated and separated, which is good practice. It does mean that we need to be sure that the inputs to our Components work as expected. Being able to test parts of our application in isolation is important because then we know that when all the various parts are passed into the components, they work. This is why Test-driven development (TDD) within Angular is important.

One of the reasons Angular is so popular in the enterprise software space is because Angular insists on **Unit Tests** being written, and well-tested software is important in the enterprise space. But before we look into how Angular makes testing so easy, we need to understand what is meant by testing and writing tests for our code. Once we understand what testing is, then we can see the ways Angular makes testing a core part of the code we write.

The first thing we need to look at is how tests are written, and for that, we need to start with Jasmine.

## Jasmine: Behavior-Driven JavaScript

**Jasmine** is a Behavior-Driven framework for writing code that tests JavaScript.

While we are writing TypeScript in Angular, we still use Jasmine to write the tests for our Angular code.

📝 **Note:** **Behavior-Driven Development (BDD)** combines TDD techniques with the domain-specific design so that both developers and the business can share how the code being written should behave. Unlike pure TDD, where tests are written just to test the code, with BDD, tests can be written to check that the code works as expected and fulfills the businesses’ needs.

[Here](https://jasmine.github.io/) is the official Jasmine website, where you’ll find a lot of information about Jasmine and writing tests. It is well worth going through this site.

The Jasmine framework was originally created by a company called [Pivotal Labs](https://run.pivotal.io/) to write BDD tests for their applications. They thankfully saw the great benefits that Jasmine can provide and released it to everyone to use in their own projects, and it soon became the default choice for writing tests.

## Angular and Jasmine

In Angular, Jasmine was soon used to write Unit Tests. This has been carried on and improved in Angular with the Jasmine framework being installed when we create a new Angular application through the Angular CLI.

📝 **Note:** While Jasmine is a very popular choice for writing unit tests and is installed by default when a new application is generated, there are other frameworks available for writing tests in Angular – frameworks such as [Mocha](https://mochajs.org/), [Chai](https://www.chaijs.com/), and [Jest](https://jestjs.io/) – and we can replace Jasmine with these other frameworks if we find that a team wants to use a different framework. These other approaches can be set up using NPM as part of generating a new Angular application. We can also tell the Angular CLI to generate a new application and not install Jasmine as part of the setup.

With Jasmine being so popular and Angular’s long history with Jasmine from back in the AngularJS days, we will continue using Jasmine when writing tests for this course.

# Jasmine in Action

Let's explore how we can write tests in Jasmine with the help of an example.

**We'll cover the following**

* [Running Jasmine test](https://www.educative.io/courses/getting-started-with-angular/RMZVj964x9V#Running-Jasmine-test)
* [The test suite](https://www.educative.io/courses/getting-started-with-angular/RMZVj964x9V#The-test-suite)
  + [Example](https://www.educative.io/courses/getting-started-with-angular/RMZVj964x9V#Example)
  + [Syntax](https://www.educative.io/courses/getting-started-with-angular/RMZVj964x9V#Syntax)
* [The test spec](https://www.educative.io/courses/getting-started-with-angular/RMZVj964x9V#The-test-spec)
  + [Example](https://www.educative.io/courses/getting-started-with-angular/RMZVj964x9V#Example)
  + [Syntax](https://www.educative.io/courses/getting-started-with-angular/RMZVj964x9V#Syntax)
* [The Expectation expression](https://www.educative.io/courses/getting-started-with-angular/RMZVj964x9V#The-Expectation-expression)
  + [Example](https://www.educative.io/courses/getting-started-with-angular/RMZVj964x9V#Example)
  + [Syntax](https://www.educative.io/courses/getting-started-with-angular/RMZVj964x9V#Syntax)
* [The Matcher expression](https://www.educative.io/courses/getting-started-with-angular/RMZVj964x9V#The-Matcher-expression)
  + [The not matcher](https://www.educative.io/courses/getting-started-with-angular/RMZVj964x9V#The-not-matcher)

## Running Jasmine test

The first thing we need to do is have an example piece of code to test.

Say we have a function that returns the name of the book in the browser. This function would look like this:

function writeTitle() {  
   return 'Getting Started With Angular';  
}

A very simple function, but it will allow us to write a simple test. The test will look like this:

describe('writeTitle function', () => {  
       it('returns Getting Started With Angular', () => {  
           expect(writeTitle()).toEqual(  
               'Getting Started With Angular'  
           );  
       });  
});

So, this is our first test. It is a very simple example, but it does have a lot going on. There are a number of features of the Jasmine framework being used in this example. Let’s take a minute to explore what these features are.

📝 **Note:** Press the **RUN** button to run the test given below.

###### /

Title.js

spec

TitleSpec.js

support

**TitleSpec.js**

1

2

3

4

5

6

7

8

9

10

11

12

13

var app = require("../Title.js");

// Test suite

describe('writeTitle function', () => {

       // Test spec

       it('returns Getting Started With Angular', () => {

           // Expectation expression

           expect(app.writeTitle()).toEqual(

               'Getting Started With Angular'

           );

       });

});





Run

Save

Reset

Jasmine test

You should see the following in the output:

1 spec, 0 failures  
Finished in 0.009 seconds

As you can see from the output, there is 1 test spec and 0 failures. If there was a failing test, the output would show it.

📝 **Note:** Try changing the title returned by the function in the above code and see how the test fails.

Show Solution

## The test suite

The first line of this test, which starts with the describe() statement, is called a test suite.

A **test suite** is a function that sets out a series of test specs related to either a piece of code or a section of functionality.

### Example

You could create a test suite, which contains a series of test specs all about one function. These test specs could be set up to test various scenarios of functionality that this code should be able to do. Or, we could set up a test suite that contains a series of test specs designed to test a piece of functionality with our application.

### Syntax

The test suite takes in two arguments: a string and a function.

describe('title', () => { } );

The **string** can be a title that describes what the test suite is testing. In our example, we have a title called the writeTitle function. This title is useful for two reasons. Firstly, it allows us to see what this test suite is designed to be testing, which is very helpful if we are working on a test suite written by another developer. And the second reason this is very useful is that when the tests are run in a test runner (which we will be looking at later in this chapter), the output of the test runner uses the title to show both what tests have passed and, more importantly, what tests have failed. When a test fails, we can see the title of the test suite the failed test belongs to.

The **second argument** of the test suite function is another function, which is run when the test suite is run by the Test Runner. This function contains all the test specs for this suite.

## The test spec

The second part of this example is the test spec, which starts with the it() statement.

The **test spec** contains one or more expectations, where we set out what we expect the code being tested to be able to do.

### Example

In the previous example, we have one single expect statement that calls/runs the writeTest() function we are testing and sets out what we expect to get returned from the function being tested. We can have a number of these expected statements within a test spec, so we set out our code’s various expectations. All these expectations are contained within the one test spec.

### Syntax

Like the test suite, a test spec takes two arguments: a title and a function.

it('title', () => { });

The **title** allows us to describe what we expect the function under test to do. In this example, we are saying that it should return Getting Started With Angular. This title is helpful, as it tells us what the test is expecting of the code under test, and in the test runner output, we can see what tests have failed.

The **second argument** of the test spec is the function. This is, again, run like the test suite, but unlike the test suite, there is only one function for a test spec. It does not contain multiple functions, only expectations.

## The Expectation expression

Within our test spec, we have an Expectation expression, which is the line that starts with the expect() keyword.

In the Expectation expression, we simply describe what we expect the function under test to do.

### Example

In our example, we expect that the writeTitle() function will return 'Getting Started With Angular'. If the writeTitle() function was to return 'Getting Started With Angular 15', then the test has failed because it does not meet our expectations.

📝 **Note:** The term function under test, which we have been using throughout this chapter, is how we describe the function being tested. In this current example, the function under test refers to the writeTitle() function.

### Syntax

There are two parts to the Expectation expression. The first is the **expect** statement, and the second is the matcher, which in our example is the toEqual() statement. The Expectation expression and a matcher are combined to make a complete expression of what we want the function under test to do.

expect(functionName).toEqual(value);

Within the Test Spec, we can have multiple Expectation expressions. For example, we could add more expect statements to check other scenarios that we may or may not want our function to do:

###### /

Title.js

spec

TitleSpec.js

support

**TitleSpec.js**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

var app = require("../Title.js");

// Test suite

describe('writeTitle function', () => {

   // Test spec

   it('should return the title', () => {

       // First Expectation expression

       expect(app.writeTitle()).toEqual(

           'Getting Started With Angular'

       );

       // Second Expectation expression

       expect(app.writeTitle()).not.toEqual(

           'Getting Started With React'

       );

    });

});





Run

Save

Reset

Running Test Spec with multiple expectations

Here, we have expanded on the test spec to have another Expectation expression that says that the returned string from our writeTitle() function is not equal to Getting Started With React. Now we are checking for two scenarios: what our function under test should do and what it shouldn’t do.

## The Matcher expression

So far, we’ve seen just one Matcher expression, the toEqual() statement. There are actually a load more matcher statements we can access from Jasmine. Here’s a list of few of them:

* toContain()
* toThrow()
* not
* toBeGreaterThan()
* toBeTruthy()
* toContain()
* toEqual()
* toHaveBeenCalled()
* toMatch()
* withContext()

That’s a lot of possible matchers we can use with our Expectation expressions. To see how these matchers work and the arguments they take, check out the [official Jasmine API docs](https://jasmine.github.io/api/edge/matchers.html), which goes through this list and shows examples of how they can be used.

### The not matcher

You may have noticed that the not matcher in the list doesn’t look the same as the others. It doesn’t have the brackets indicating that it can take in a set of arguments. This is because this matcher is used to switch the expectation expression around.

In our example, we have one expect expression that says what our function should do. Then, we use the not match to create another one that says what the function should not do. We can use the not matcher with all the other matchers from the previous list to switch the expression around so we can set expectations of what a function under test should not do as well as what it should do. This builds up a test spec that sets out clearly what a piece of code can and cannot do in order to pass the test we are writing.

# Setup and Teardown of Tests

Let's explore how we can use the setup and teardown function to initialize something within the test spec.

**We'll cover the following**

* [Teardown functions](https://www.educative.io/courses/getting-started-with-angular/xoDkwY81E7E#Teardown-functions)
  + [Example of beforeAll() and afterAll() functions](https://www.educative.io/courses/getting-started-with-angular/xoDkwY81E7E#Example-of-beforeAll()-and-afterAll()-functions)
  + [Explanation](https://www.educative.io/courses/getting-started-with-angular/xoDkwY81E7E#Explanation)
  + [Why use setup and teardown functions?](https://www.educative.io/courses/getting-started-with-angular/xoDkwY81E7E#Why-use-setup-and-teardown-functions?)

## Teardown functions

When writing tests, sometimes we need to set up some things before running a test spec.

The type of things we may need to initialize before running a test can include:

* Creating mock data
* Setting some global variables
* Creating mock services needed to run the function under test.

All this can be handled as part of the setup of a test, but we can also clear these mock services and data as part of the teardown of a test suite. To do this, Jasmine provides us with four functions we can use in a test suite, which are set out in the following list:

| **Function** | **Purpose** |
| --- | --- |
| **beforeAll()** | This function is run once, before all the test specs within a suite |
| **beforeEach()** | This function is run before each test spec is run |
| **afterAll()** | This function is run once, after all the test specs have been run |
| **afterEach()** | This function is run after each test spec has run |

As you can see, there are two types of setup and teardown functions. There are ones that are run just once when a test suite is run and those that are run once for each test spec.

### Example of beforeAll() and afterAll() functions

Let’s have a look at how the beforeAll() and afterAll() functions could be used:

📝 **Note:** Press the **RUN** button, and see what happens!

###### /

Title.js

spec

TitleSpec.js

support

**TitleSpec.js**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

var app = require("../Title.js");

describe('writeTitle function', () => {

   var bookTitle = '';

   beforeAll(() => {

           bookTitle = 'Getting Started With Angular';

    });

    it('should return the title', () => {

        expect(app.writeTitle()).toEqual(bookTitle);

    });

    afterAll(() => {

        bookTitle = '';

    });

});





Run

Save

Reset

Setup and Teardown function

You should see the following in the output:

1 spec, 0 failures  
Finished in 0.013 seconds

As you can see from the output, there is 1 test spec and 0 failures.

### Explanation

In this example, we’re creating a local bookTitle variable, which is used as the argument is passed into the toEqual() matcher expression. In the beforeAll() function, we are setting this property to Getting Started With Angular so that the bookTitle property is set before all of our tests (which we only have one of in this example test spec) are run. Then, after all the tests have run, we reset the local bookTitle property back to an empty property.

So, you can see that in beforeAll() we are setting up the state for the tests in the test suite, then after they have all run, we reset this state data back to its original state.

### Why use setup and teardown functions?

Why would we want to do this? Well, say, for example, we want to make sure that our test data is reset in case one of our tests updates or amends this state data, and we want to have this data reset for when the tests run again. The setup and teardown functions can be used to create and clear up mock data needed for the tests.

These functions really come into their own when we need mock data and services needed for our tests to run, which we will see later when we explore writing more complex tests.

# The Karma Test Runner

Let's explore how we can use the Karma test runner to run test specs and test suites.

**We'll cover the following**

* [What is Karma?](https://www.educative.io/courses/getting-started-with-angular/gk3r1vxPDlD#What-is-Karma?)
* [How Karma is used](https://www.educative.io/courses/getting-started-with-angular/gk3r1vxPDlD#How-Karma-is-used)
* [Why use Karma?](https://www.educative.io/courses/getting-started-with-angular/gk3r1vxPDlD#Why-use-Karma?)

## What is Karma?

In Angular, along with having Jasmine, there is the **Karma test runner**, which is part of the testing process of Angular. Again, like Jasmine, Karma is installed as part of a new Angular application (we can also create a new Angular application without installing Karma).

📝 **Note:** When we say running our tests, this means we go through all the written test suite files and trigger the describe() functions within each test suite, which in turn runs all the test specs to test the code we’ve created to meet our expectations.

Karma aims to give developers immediate feedback on the tests we’ve written, so we know exactly where and when the code we’re writing has not passed a test. While by default we use Jasmine and Karma together, Karma will run other testing frameworks such as Jest, Mocha, and QUnit. Meaning that while we are using Jasmine in this course, if you work with a team that uses Jest instead of Jasmine, the use of Karma is still the same.

## How Karma is used

Karma is a command-line tool that starts up a local webserver (similar to the Angular CLI when we run ng serve).

This local web server is used to execute the application code against our tests, and then Karma will give immediate feedback on any passing or failing tests in either the command line or a local browser as an HTML-based report, which you can see an example of here:

📝 **Note:** Press the **RUN** button to run the tests. Once the test runs, click on the URL given after Your app can be found at, and see what happens!

###### /

e2e

src

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**karma.conf.js**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

// Karma configuration file, see link for more information

// https://karma-runner.github.io/1.0/config/configuration-file.html

module.exports = function (config) {

  config.set({

    basePath: '',

    frameworks: ['jasmine', '@angular-devkit/build-angular'],

    plugins: [

      require('karma-jasmine'),

      require('karma-chrome-launcher'),

      require('karma-jasmine-html-reporter'),

      require('karma-coverage-istanbul-reporter'),

      require('@angular-devkit/build-angular/plugins/karma')

    ],

    client: {

      clearContext: false // leave Jasmine Spec Runner output visible in browser

    },

    coverageIstanbulReporter: {

      dir: require('path').join(\_\_dirname, './coverage/Client-Contacts-Manager-Angular'),

      reports: ['html', 'lcovonly', 'text-summary'],

      fixWebpackSourcePaths: true

    },

    reporters: ['progress', 'kjhtml'],

    port: 9876,

    colors: true,

    logLevel: config.LOG\_INFO,

    autoWatch: true,

    browsers: ['Chrome'],

    singleRun: false,

    restartOnFileChange: true

  });

};





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/debug.html>

Run unit tests through Karma

As you can see from this report, there are **3 test specs and 0 failures**. If there was a failing test, the report would show it.

ng test output

Karma is running the local web server, running the Tests against the application and generating the test report in the browser. But as we said earlier, Karma is a command-line tool. So we can also see the report in the Terminal.

When the test fails, it shows which test has failed (we’re using the title of test spec in the output of the report) and even what line of code the failed test can be found at.

## Why use Karma?

Karma can also continuously watch all our application files, and if there is a change, it can re-run all the tests to see if the change we’ve made has fixed the code being tested. If the change has fixed the failing code, the generated report will show a green line indicating that all the tests have passed.

As we continue developing our application, we can keep the Karma web server running and monitoring our application to keep informing us if the code we’re writing is passing the tests we’ve written. This part of the continuous development is a key part of Test Driven Development where we write the tests defining what our code should do then we write the code to pass the tests.

# Taking a Test Driven Development Approach

Learn what a test-driven development (TDD) approach is, the rules of the TDD approach, and why it should be followed while writing tests for Angular applications

**We'll cover the following**

* [What is TDD?](https://www.educative.io/courses/getting-started-with-angular/3YKYVoq0wkn#What-is-TDD?)
* [The rules of test driven development](https://www.educative.io/courses/getting-started-with-angular/3YKYVoq0wkn#The-rules-of-test-driven-development)
* [Simplicity criteria by Kent Beck](https://www.educative.io/courses/getting-started-with-angular/3YKYVoq0wkn#Simplicity-criteria-by-Kent-Beck)
* [Why use TDD in Angular?](https://www.educative.io/courses/getting-started-with-angular/3YKYVoq0wkn#Why-use-TDD-in-Angular?)

## What is TDD?

Before moving onto seeing how tests are set up within an Angular application, we need to understand what is meant by taking a Test Driven Development or TDD approach to writing code.

TDD has been around since the 1970s. The exact date when it started being used is hard to say since the practice of TDD has evolved over time as testing has grown in various languages and frameworks. And as testing became more and more important, a set of rules were agreed on for an approach on how we should write code. These rules became the base for the TDD approach.

According to the [Agile Alliance](https://www.agilealliance.org/), which is designed to promote Agile Development practices, and which TDD is part of (along with Sprints, Stand-ups, and pair programming), the definition of TDD is:

**Test-driven development** refers to a style of programming in which three activities are tightly interwoven: coding, testing (in the form of writing unit-tests), and design (in the form of refactoring).

## The rules of test driven development

As part of this definition, the Agile Alliance has a set of rules that describes TDD:

* Write a single unit test describing an aspect of the program (our test specs)
* Run the test, which should fail because the program lacks the feature
* Write ‘just enough’ code to make the test pass
* Refactor the code until it conforms to the simplicity criteria
* Repeat, accumulating unit tests over time

Test-Driven Development

So, according to these rules, when taking a TDD approach to our development, we should start writing a test spec that sets out what we expect a piece of functionality to do, then run this test, which will fail because we haven’t written the implementation code yet. Then we write the simplest code we can to make the test pass. Once this code is in place, we refactor this code, making sure it still passes the test. Continuing this process throughout the development of our application. By taking this approach, we have a growing set of tests that confirm all the code of our application passes the tests.

## Simplicity criteria by Kent Beck

The simplicity criteria have been set out by **Kent Beck**, the creator of extreme programming, to judge whether some code is simple enough:

* The code is verified by automated unit tests, and all tests must pass
* The code contains no duplication
* The code expresses separately each distinct idea or responsibility
* The code is composed of the minimum number of components (classes, methods, and lines) compatible with the first three criteria

## Why use TDD in Angular?

By taking a TDD approach in Angular, we would be writing our tests before writing the code of our Component classes and services. This approach brings a lot of benefits, including knowing that tests cover all your code. So if we refactor our code or add a new feature, we will be aware of any new bugs that may surface in our code over time.

TDD, while a great approach to development, is not a requirement of writing tests in Angular. It is worth knowing the strategies of TDD, though, as it keeps our tests as small as possible and tests the public interfaces into our functions, even if we don’t follow the approach of writing tests before we write our code.

# Karma Settings in Angular

Learn to set up testing in Angular as well as the properties in the angular.json file and the karma.conf.js file.

**We'll cover the following**

* [Setup testing in an Angular application](https://www.educative.io/courses/getting-started-with-angular/3j8AlG6X6P4#Setup-testing-in-an-Angular-application)
* [Test settings in angular.json](https://www.educative.io/courses/getting-started-with-angular/3j8AlG6X6P4#Test-settings-in-angular.json)
  + [The builder property](https://www.educative.io/courses/getting-started-with-angular/3j8AlG6X6P4#The-builder-property)
  + [The karmaConfig property](https://www.educative.io/courses/getting-started-with-angular/3j8AlG6X6P4#The-karmaConfig-property)
* [Test settings in karma.conf.js file](https://www.educative.io/courses/getting-started-with-angular/3j8AlG6X6P4#Test-settings-in-karma.conf.js-file)

## Setup testing in an Angular application

From what we’ve learned about Jasmine and Karma, we could set up testing within our Angular applications ourselves. All we need to do is:

* Install Karma
* Set the configuration file telling Karma to run
* Continuously watch for changes to our files in order to trigger the re-running of tests.
* Install the Jasmine framework into our application via NPM.

📝 **Note:** The [Karma website](http://karma-runner.github.io/5.2/intro/installation.html) has a great tutorial on how to set up Karma for a project, and you can find details on adding Jasmine from the [Jasmine website](https://jasmine.github.io/pages/getting_started.html).

Thankfully for us, all this has been handled through the Angular CLI. When we create a new project, the Angular CLI downloads and installs all, we need for testing our applications.

📝 **Note:** If you want to create a new Angular application without the tests and test files, you can by adding the --skipTests=true argument to the ng new command like this:

ng new MyTestFreeApp --skipTests=true

In an Angular project, there are two separate files, which manage tests. They are the angular.json file and the karma.conf.js file.

## Test settings in angular.json

In the angular.json config file, which is the main config file of an Angular application, there is a section where the test command of the Angular CLI settings can be found:

 "test": {  
  "builder": "@angular-devkit/build-angular:karma",  
  "options": {  
    "main": "src/test.ts",  
    "polyfills": "src/polyfills.ts",  
    "tsConfig": "tsconfig.spec.json",  
    "karmaConfig": "karma.conf.js",  
    "assets": [  
      "src/favicon.ico",  
      "src/assets"  
    ],  
    "styles": [  
      "src/styles.scss"  
    ],  
    "scripts": []  
  }  
},

This section from the angular.json file provides all the details needed when our tests are run. There are two sections that we are looking at here:

* The builder property
* The karmaConfig property.

### The builder property

The builder property tells the Angular CLI what building tool it should use when running the test command, which in this example is using the default builder of Karma. This could be different if we’re using another test runner.

### The karmaConfig property

The karmaConfig property tells the CLI where the karma.conf.js file can be found. This JavaScript file is the configuration file for the Karma test runner.

## Test settings in karma.conf.js file

If we look at this file, we can see a number of settings that tell Karma how to run:

###### /

e2e

src

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**karma.conf.js**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

// Karma configuration file, see link for more information

// https://karma-runner.github.io/1.0/config/configuration-file.html

module.exports = function (config) {

  config.set({

    basePath: '',

    frameworks: ['jasmine', '@angular-devkit/build-angular'],

    plugins: [

      require('karma-jasmine'),

      require('karma-chrome-launcher'),

      require('karma-jasmine-html-reporter'),

      require('karma-coverage-istanbul-reporter'),

      require('@angular-devkit/build-angular/plugins/karma')

    ],

    client: {

      clearContext: false // leave Jasmine Spec Runner output visible in browser

    },

    coverageIstanbulReporter: {

      dir: require('path').join(\_\_dirname, './coverage/Client-Contacts-Manager-Angular'),

      reports: ['html', 'lcovonly', 'text-summary'],

      fixWebpackSourcePaths: true

    },

    reporters: ['progress', 'kjhtml'],

    port: 9876,

    colors: true,

    logLevel: config.LOG\_INFO,

    autoWatch: true,

    browsers: ['Chrome'],

    singleRun: false,

    restartOnFileChange: true

  });

};





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/debug.html>

Karma config file

This Karma config file, which is from a standard Angular application generated from the Angular CLI, has a number of properties. Let’s take a quick look at what these properties are:

| **Properties** | **Purpose** |
| --- | --- |
| **basePath** | Used to resolve the path to any file. |
| **frameworks** | An array of the testing framework being used. In this case, Jasmine. |
| **plugins** | An array of plugins needed. In this example we have plugins that Karma needs to run the app in the browser, link with Jasmine, and generate the report in the browser. |
| **client** | An object with arguments that are passed to the browser. Here, we are passing the clearContext, which clears the context of the client forcing a clean run every time the tests are run. |
| **coverageInstanbulReporter** | This is an object with settings for the HTML report that Karma uses to show the outcome of the test. |
| **reporters** | A list of the reporters that should be used. Here, we are having it show us the report on the progress of the tests being run. As tests are being run, we can see how they are progressing. |
| **logLevel** | This is the level of the information returned to log as tests are being run. By default, we get the general information level, but we could change this to show just errors or full debug information. |
| **autoWatch** | This is the setting that tells Karma to watch for changes to our files and if there have been any changes, run the tests. |
| **browsers** | This is the browser we are using to run the app. We can run tests in more than one browser. |
| **singleRun** | This is used if Karma should run all the tests in all the lists of browsers continuously as we work. |

There is a huge set of configuration settings for Karma. As well as these, it’s worth familiarizing yourself with them by going to the Configuration settings page of the [Karma website](http://karma-runner.github.io/5.2/config/configuration-file.html).

Now we know what these settings are, we can amend them to our needs if we want. For example, we could change logLevel to config.LOG\_ERROR to just return information on when a test fails/errors to the console or report. Karma is very flexible, and even though Karma has been installed and set up as part of the build of a new Angular application by the CLI, we can still tweak the settings for our own needs.

# Running Tests Using the Angular CLI

Let's explore how we can run tests using the Angular CLI.

**We'll cover the following**

* [Command to run tests](https://www.educative.io/courses/getting-started-with-angular/YQpK935jmGO#Command-to-run-tests)
  + [Test command arguments](https://www.educative.io/courses/getting-started-with-angular/YQpK935jmGO#Test-command-arguments)
    - [--codeCoverage argument](https://www.educative.io/courses/getting-started-with-angular/YQpK935jmGO#--codeCoverage-argument)
    - [--watch argument](https://www.educative.io/courses/getting-started-with-angular/YQpK935jmGO#--watch-argument)

## Command to run tests

To run the tests within an Angular application, we can simply use a single command to the Angular CLI:

ng test

This will begin the process of starting Karma, launching the browser, and finding and running all the test specs we have in our code base. Then, within the same Terminal window, we will see the outcome of the tests and if they have passed or failed.

📝 **Note:** Press the **RUN** button to run the ng test command. Once the test runs, click on the URL given after Your app can be found at, and see what happens!

###### /

e2e

src

app

app-routing.module.ts

app.component.html

app.component.sass

app.component.spec.ts

app.component.ts

app.module.ts

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.sass

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**app.component.spec.ts**

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

  beforeEach(async () => {

    await TestBed.configureTestingModule({

      imports: [

        RouterTestingModule

      ],

      declarations: [

        AppComponent

      ],

    }).compileComponents();

  });

  it('should create the app', () => {

    const fixture = TestBed.createComponent(AppComponent);

    const app = fixture.componentInstance;

    expect(app).toBeTruthy();

  });

  it(`should have as title 'Client-Contacts-Manager-Angular'`, () => {

    const fixture = TestBed.createComponent(AppComponent);

    const app = fixture.componentInstance;

    expect(app.title).toEqual('Client-Contacts-Manager-Angular');

  });

  it('should render title', () => {

    const fixture = TestBed.createComponent(AppComponent);

    fixture.detectChanges();

    const compiled = fixture.nativeElement;

    expect(compiled.querySelector('.content span').textContent).toContain('Client-Contacts-Manager-Angular app is running!');

  });

});





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/debug.html>

Run unit tests

Now after the tests have run, we can see the output of the tests. As you can see from this output, 3 tests have been passed.

### Test command arguments

The ng test command, like so many of the other Angular CLI commands, has a set of arguments that can be passed along with this command. A couple of arguments that are extremely useful are:

* --codeCoverage argument
* --watch argument

#### --codeCoverage argument

This can be true or false. If true, it will generate a report telling you how much of your code is covered by tests. So you can see if there are any areas that need more tests.

#### --watch argument

This also can be true or false. If true, it runs the ng test whenever a file is changed, so you have this set to false in the Karma config file. But if you want to keep the tests running when you are working on a complex section of the application, you can start the ng test command with this argument to keep tests running.

To see what other arguments are available for the ng test command, check out the [official documentation](https://angular.io/cli/test).

# Writing Tests in Angular

Learn to generate tests for components using the Angular CLI as well as the properties in the Test.ts and the auto-generated spec file.

**We'll cover the following**

* [Angular CLI and test spec file](https://www.educative.io/courses/getting-started-with-angular/gkypR080zX6#Angular-CLI-and-test-spec-file)
* [The test.ts file](https://www.educative.io/courses/getting-started-with-angular/gkypR080zX6#The-test.ts-file)
  + [Initializing Angular’s testing environment](https://www.educative.io/courses/getting-started-with-angular/gkypR080zX6#Initializing-Angular%E2%80%99s-testing-environment)
  + [Initializing context property](https://www.educative.io/courses/getting-started-with-angular/gkypR080zX6#Initializing-context-property)
* [The auto-generated test spec file](https://www.educative.io/courses/getting-started-with-angular/gkypR080zX6#The-auto-generated-test-spec-file)
  + [Expectation expression](https://www.educative.io/courses/getting-started-with-angular/gkypR080zX6#Expectation-expression)
  + [describe() function](https://www.educative.io/courses/getting-started-with-angular/gkypR080zX6#describe()-function)
    - [The component property](https://www.educative.io/courses/getting-started-with-angular/gkypR080zX6#The-component-property)
    - [The fixture property](https://www.educative.io/courses/getting-started-with-angular/gkypR080zX6#The-fixture-property)
  + [beforeEach( ) functions](https://www.educative.io/courses/getting-started-with-angular/gkypR080zX6#beforeEach(-)-functions)

## Angular CLI and test spec file

Not only does the Angular CLI install and set up Karma, as well as have a command to run all our tests, it also automatically generates a test spec file for us every time we use the CLI to generate a Component or service.

You may have seen this when we’ve used the ng generate command in earlier chapters, where we’ve created either a new Component or a service. The list of files being generated has always included a spec.ts file.

So, if we were to use the Angular CLI to generate a new component for use, we would use this command:

ng generate component my-comp

📝 **Note:** Click the terminal window and see what happens!

**Terminal 1**

Terminal

**Click to Connect...**

****

This would create four separate files:

* my-comp.component.html: The template file of our component
* my-comp.component.scss: The scss for the component
* my-comp.component.ts: The TypeScript class of the component
* my-comp.component.spec.ts: The test file for the component

The file we’re interested in here is the my-comp.component.spec.ts file, which is our automatically generated test file.

Every time one of these spec.ts files is generated, Karma is automatically aware of the file, so we do not need to update a config file with the names of any new Spec files. Angular makes Karma aware of these files through a separate TypeScript file called test.ts.

## The test.ts file

In this test.ts file, which can be found in the src folder, Angular’s testing environment is set up using the BrowserTestingDynamicModule of Angular. If we look at the test.ts, we can see how this is initialized:

import 'zone.js/dist/zone-testing';  
import { getTestBed } from '@angular/core/testing';  
import {  
  BrowserDynamicTestingModule,  
  platformBrowserDynamicTesting  
} from '@angular/platform-browser-dynamic/testing';  
  
declare const require: {  
  context(path: string, deep?: boolean, filter?: RegExp): {  
    keys(): string[];  
    <T>(id: string): T;  
  };  
};  
  
// First, initialize the Angular testing environment.  
getTestBed().initTestEnvironment(  
  BrowserDynamicTestingModule,  
  platformBrowserDynamicTesting()  
);  
// Then we find all the tests.  
const context = require.context('./', true, /\.spec\.ts$/);  
// And load the modules.  
context.keys().map(context);

This file is performing two main actions:

* Initializing Angular’s testing environment
* Initializing context property

### Initializing Angular’s testing environment

Initializing Angular’s testing environment, which is the environment Angular runs in when running our tests, is different from when Angular is running the application in the browser.

### Initializing context property

Initializing context property, which uses a regular expression to find all the spec.ts files in our codebase. It creates a map of all these spec files, which is available to Karma. So, when the Karma test runner starts and runs, it uses this mapping to go through all the available spec files.

## The auto-generated test spec file

If we open one of these spec files for a component and see what that CLI has created for us, it’ll look like this:

import { ComponentFixture, TestBed } from '@angular/core/testing';  
  
import { MyCompComponent } from './my-comp.component';  
  
describe('MyCompComponent', () => {  
  let component: MyCompComponent;  
  let fixture: ComponentFixture<MyCompComponent>;  
  
  beforeEach(async () => {  
    await TestBed.configureTestingModule({  
      declarations: [ MyCompComponent ]  
    })  
    .compileComponents();  
  });  
  
  beforeEach(() => {  
    fixture = TestBed.createComponent(MyCompComponent);  
    component = fixture.componentInstance;  
    fixture.detectChanges();  
  });  
  
  it('should create', () => {  
    expect(component).toBeTruthy();  
  });  
});

This is a test spec file of the MyComp component we have created above. Even though this is a generated file, there is still a bit going on here.

### Expectation expression

The first thing to know is that while this is a TypeScript file, it’s still using the Jasmine test framework. From what we’ve already learned about Jasmine, we can see that there is an expectation expression and a matcher from Jasmine used in the single test we have in this file:

it('should create', () => {  
   expect(component).toBeTruthy();  
});

### describe() function

The file starts with a describe() function, as all test files do. In this, we have the title MyCompComponent, and the CLI uses the name of the component to generate this title. Then we have two local properties:

* component
* fixture

#### The component property

The component property is a reference to the component TypeScript file, so we have access to the component’s properties through this local property.

#### The fixture property

The fixture property creates a ComponentFixture object, which we will explore in a bit.

### beforeEach( ) functions

Then we have two beforeEach() functions. We can have more than that if we choose. In the first of these functions, the TestBed class is being provided with a declarations array, consisting of the one component file.

In the second beforeEach() function, we are setting the TestBed class to create the component being tested and set this as the fixture property, then getting an instance of the fixture and setting that to our local component property. Finally, we are calling the **fixture** property’s detectChanges() function. All this is to set up how Angular knows about the component being tested and making it available to Angular’s testing classes.

To understand what is happening here, we need to look at the [TestBed class and the ComponentFixture class](https://www.educative.io/collection/page/10370001/4603693004488704/4960833024032768" \t "_blank) Angular provides.

# The TestBed Class

Learn what TestBed class is, how we can inject services into the TestBed class, and what benefits it brings to our application.

**We'll cover the following**

* [What is the TestBed class?](https://www.educative.io/courses/getting-started-with-angular/N8kOAn41Jkz#What-is-the-TestBed-class?)
  + [First beforeEach() function](https://www.educative.io/courses/getting-started-with-angular/N8kOAn41Jkz#First-beforeEach()-function)
  + [Second beforeEach() function](https://www.educative.io/courses/getting-started-with-angular/N8kOAn41Jkz#Second-beforeEach()-function)
* [Injecting a service using the TestBed](https://www.educative.io/courses/getting-started-with-angular/N8kOAn41Jkz#Injecting-a-service-using-the-TestBed)
* [Create a spy](https://www.educative.io/courses/getting-started-with-angular/N8kOAn41Jkz#Create-a-spy)
  + [A brief word about spies](https://www.educative.io/courses/getting-started-with-angular/N8kOAn41Jkz#A-brief-word-about-spies)
* [Mocks in Jasmine](https://www.educative.io/courses/getting-started-with-angular/N8kOAn41Jkz#Mocks-in-Jasmine)
  + [Why use mocks?](https://www.educative.io/courses/getting-started-with-angular/N8kOAn41Jkz#Why-use-mocks?)
  + [Example](https://www.educative.io/courses/getting-started-with-angular/N8kOAn41Jkz#Example)
* [Stubs in tests](https://www.educative.io/courses/getting-started-with-angular/N8kOAn41Jkz#Stubs-in-tests)
  + [Example](https://www.educative.io/courses/getting-started-with-angular/N8kOAn41Jkz#Example)
* [Benefits of the TestBed class](https://www.educative.io/courses/getting-started-with-angular/N8kOAn41Jkz#Benefits-of-the-TestBed-class)

## What is the TestBed class?

The Angular TestBed class helps us by providing an environment for testing our application. It also provides an API for making our components and services available to the Unit Tests.

In AngularJS, we didn’t have the TestBed class, which made it far more complex to access the Controllers and services in the Unit Tests. Thankfully, with the TestBed class, it is far easier to access our components and services in order to test them.

Going back to our example spec file, let’s see how the TestBed class is being used.

import { ComponentFixture, TestBed } from '@angular/core/testing';  
  
import { MyCompComponent } from './my-comp.component';  
  
describe('MyCompComponent', () => {  
  let component: MyCompComponent;  
  let fixture: ComponentFixture<MyCompComponent>;  
  
  beforeEach(async () => {  
    await TestBed.configureTestingModule({  
      declarations: [ MyCompComponent ]  
    })  
    .compileComponents();  
  });  
  
  beforeEach(() => {  
    fixture = TestBed.createComponent(MyCompComponent);  
    component = fixture.componentInstance;  
    fixture.detectChanges();  
  });  
  
  it('should create', () => {  
    expect(component).toBeTruthy();  
  });  
});

In the two beforeEach() functions, which, as we know, are run before each test spec within the file, we are using the TestBed to make the component we’re testing accessible.

### First beforeEach() function

In the first beforeEach(), the TestBed is being configured through calling its configureTestingModule() method. This tells the test environment’s module of all the declarations it needs to know. This is similar to how the main app.module.ts is set up. If we go back to what we learned about the main NgModule class in [Chapter 6: NgModule](https://www.educative.io/collection/page/10370001/4603693004488704/4533414119079936), the same set of arrays (declarations, imports, providers, and so on) that we pass into the app.module.ts file can be passed into the Testing environment module via the TestBed class. When we look at some examples of tests, you’ll see this list grow, depending on what is being tested.

### Second beforeEach() function

In the second beforeEach() method, we create this **fixture property**, which is a wrapper for the component being tested and the template of the component. This is helpful for accessing the template’s HTML if we want to test if a value in the template has been updated.

Then we create an instance of this component based on the fixture wrapper property. The reason we do this is so that when a test is run, it has a new instance of the component from the fixture. For example, if we have one test that needs to set a property in the component to test if the property has been changed successfully, while in another test, we don’t want to change the same property.

Finally, we call the detectChanges() method of the fixture. This triggers all the lifecycle hooks a component has, which we learned about in [Chapter 5: Components, Templates, and Forms](https://www.educative.io/collection/page/10370001/4603693004488704/6414012458729472).

## Injecting a service using the TestBed

If our component under test requires service as part of the functionality of the component we’re testing, we can use the TestBed class to inject this service into our spec file.

For example, if we have a component that needs a LoginService, we can make this service available by adding it in our beforeEach() methods:

let loginService: LoginService;  
  
beforeEach(async(() => {  
    TestBed.configureTestingModule({  
        declarations: [ CompanyPageComponent ],  
        providers: [ LoginService ]  
        }).compileComponents();  
   }));  
         
beforeEach(() => {  
   fixture = TestBed.createComponent(CompanyPageComponent);  
   component = fixture.componentInstance;  
   fixture.detectChanges();  
  
   loginService = TestBed.get(LoginService);  
});

In this example, we’re creating a new local property called loginService, then adding the LoginService to the list of providers (an example of the providers’ array we pass to the TestingModule), so the TestingModule is aware of the service. Then we use the TestBed get() method to return an instance of the LoginService to our new local property.

## Create a spy

In a test spec, we can create a spy to check that a function from this LoginService has been called when we are testing our component:

it('checks that isloggedIn is called during login', () => {  
   // creates a Spy to see if the loginService isloggedIn() method has been called  
   spyOn(loginService, 'isloggedIn').and.returnValue(false);  
     
   // runs the logUsIn() method of the component  
   expect(component.logUsIn()).toBeTruthy();  
     
   // tests to see that the Service method has been called  
   expect(loginService.isloggedIn).toHaveBeenCalled();  
});

### A brief word about spies

When we looked at Jasmine earlier in the chapter, we didn’t look at what spies are, but now we have an example, we can learn a bit more about them and why we would use them.

A **spy** is part of the Jasmine testing framework, and a spy can stub out a function and keep track of any calls to this function.

This is useful for when, as the previous example shows, we have a function that is called as part of the code being tested, and we want to make sure that this function has been called. As the spy function keeps track of calls to the method being tracked, we can use matchers such as, hasBeenCalled() and hasBeenCalledWith(), to see if these methods are being invoked correctly.

In our example, we are creating a spy object by calling the spyOn() method and passing on the name of the spy object, loginService, and the name of the method we want to track calls to, which is the isLoggedIn method.

Spies are very useful, especially when we are testing interaction with services in our components. It is worth exploring how to use them by checking out the [official Jasmine documentation](https://jasmine.github.io/tutorials/your_first_suite).

## Mocks in Jasmine

Another useful feature of Jasmine are mocks. This is where we create mock versions of full classes within our test. These mock classes have the same signature as the real class they are replacing within our tests.

### Why use mocks?

Why would we do this? Well, we may be testing a complex function that has a few classes it’s dependent on. To isolate the piece of code under test, we would use mocks to create mock versions of the classes that our code is using. These mocks can be far simpler than the real classes they represent, which means we can be sure that the code being tested is not affected by any bugs that may be in the other classes. By isolating our code through mocking dependencies, we can be sure that our test is only testing the code we’re interested in.

Test individual elements in isolation

### Example

An example could be when we’re testing a piece of code that accesses some data from a database. Instead of having to create a database to test with, we could mock a data source for our code under test so we can test that it works with data retrieved from this mock data source. We could write a test that asserts that our code only accesses data successfully and does not need to be concerned with the database code, which we’re not testing at that time.

## Stubs in tests

Stubs in tests are similar to mocks. They provide us with the ability to create stubs of an interface we may need in our test.

### Example

For example, we might be testing a piece of code that loads in a dataset or a list of usernames. In order to prove this data to our code under test, we could create a ‘stub’ of this data to use when we make assertions in our tests.

Being able to write stubs means we can use different combinations of data stubbed out to see how the code being tested performs if it still works as we expect with different data sets. For example, a list of four users may be fine for a piece of code, but if we mock 50 users, will the tests still pass?

Spies, stubs, and mocks allow us to write small isolated tests where we can write a variety of tests for our codebase.

## Benefits of the TestBed class

There are a number of benefits we get from using the TestBed class in our tests:

* We can access our components far more easily than before.
* We can test the interaction between the component and the template, as the fixture gives us access to the template.
* We can make use of Dependency Injection in our test.
* We can duplicate the setup of our main NgModule in our Test Module, so we know the tests are using the same providers and declarations as the application.

The TestBed class has made writing tests in Angular far easier than in the previous version of Angular, where accessing components, templates, and services were far more complex and error-prone.

# Examples of Tests

Let's explore the common examples of tests in Angular.

**We'll cover the following**

* [Common test scenarios](https://www.educative.io/courses/getting-started-with-angular/7nrJyRz18jG#Common-test-scenarios)
* [Tests for a template of a component](https://www.educative.io/courses/getting-started-with-angular/7nrJyRz18jG#Tests-for-a-template-of-a-component)
* [Tests for @Input() and @Output() attributes](https://www.educative.io/courses/getting-started-with-angular/7nrJyRz18jG#Tests-for-@Input()-and-@Output()-attributes)
  + [Tests for @Input() attribute](https://www.educative.io/courses/getting-started-with-angular/7nrJyRz18jG#Tests-for-@Input()-attribute)
  + [Tests for @Output() attribute](https://www.educative.io/courses/getting-started-with-angular/7nrJyRz18jG#Tests-for-@Output()-attribute)
* [Tests for a service](https://www.educative.io/courses/getting-started-with-angular/7nrJyRz18jG#Tests-for-a-service)
  + [Example 1: Simple service](https://www.educative.io/courses/getting-started-with-angular/7nrJyRz18jG#Example-1:-Simple-service)
  + [Example 2: Complex service](https://www.educative.io/courses/getting-started-with-angular/7nrJyRz18jG#Example-2:-Complex-service)
    - [Step 1: Create a spy](https://www.educative.io/courses/getting-started-with-angular/7nrJyRz18jG#Step-1:-Create-a-spy)
    - [Step 2: Call the getBookTitle() method](https://www.educative.io/courses/getting-started-with-angular/7nrJyRz18jG#Step-2:-Call-the-getBookTitle()-method)
    - [Step 3: Checking error handler](https://www.educative.io/courses/getting-started-with-angular/7nrJyRz18jG#Step-3:-Checking-error-handler)

## Common test scenarios

Let’s move on to looking at some examples of the types of tests we may have in our Angular applications. While the functionality of each Angular application is different, there are some common types of tests we may write in our application.

The scenarios we’re going to look at are as follows:

* A test that accesses the template of a component
* A test that checks the @Input() and @Output() attributes of the component
* A test for a service

## Tests for a template of a component

In this example, we’re going to look at how to access elements within a template to see if there are changes to them based on functionality in the component class. Here is an example test spec file that checks for changes in the template:

import { ComponentFixture, TestBed, async } from '@angular/core/testing';  
import { By } from '@angular/platform-browser';  
import { ExampleComponent } from './example.component';  
     
describe('DomTestingComponent', () => {  
    let component: ExampleComponent;  
    let fixture: ComponentFixture<ExampleComponent>;  
  
    beforeEach(async(() => {  
        TestBed.configureTestingModule({  
            declarations: [ExampleComponent]  
        });   
    }));  
         
    beforeEach(() => {  
        fixture = TestBed.createComponent(ExampleComponent);  
        component = fixture.componentInstance;  
        fixture.detectChanges();  
    });  
         
    it('should create', () => {  
        expect(component).toBeTruthy();  
    });  
      
    it('should not have the display the users name', () => {  
        const usernameEl = fixture.debugElement.query(By.css('.username'));  
           expect(usernameEl).toBeNull();  
    });  
         
    it('should display the users name when set', () => {  
        component.username = 'Test User';  
        fixture.detectChanges();  
        fixture.whenStable().then(() => {  
  
        const usernameEl =   
        fixture.debugElement.query(By.css('.username'));  
          
        expect(usernameEl).not.toBeNull();  
       });  
    });  
});

We can access the template through the fixture, which, as we know, is set via the TestBed class generating a wrapper for the component and the template. Then, to access the value of the HTML element, we’re using CSS to find the class of the HTML element that contains the username:

const usernameEl = fixture.debugElement.query(By.css('.username'));

This line accesses the HTML element, using the By.css() function to find the matching element with the CSS class. When we have access to this HTML element, we can check to see if it’s null or not using the not.toBeNull() matcher from Jasmine (hopefully, now you can see how we are using the Jasmine framework to write these tests all within the Angular environment).

This is a fairly simple test, but it does show how HTML from the template can be accessed via the fixture returned from the TestBed. There are other methods as well as By.css() to search through the template. To find others, check the official Angular API docs and the Protractor API docs.

📝 **Note:** ***Protractor*** is an Angular framework for writing an end to end test, where we write tests that interact with the UI of an application, testing how a user would work with the application. While we will not be looking at Protractor in this course, once you feel confident with testing, it is worth looking at [Protractor](https://www.protractortest.org/#/).

Protractor does have a rich API, which we can use in our tests to inspect a component’s template.

## Tests for @Input() and @Output() attributes

Another common scenario is to test the @Input() and @Output() attributes of a component. Let’s start by looking at how we check that an @Input() of a component is set correctly.

### Tests for @Input() attribute

Continuing from our first example, if the component has an @Input() called age, in order to check that age is being set correctly, we could write a simple test like this:

it('should correctly display the @Input value for age', () => {  
   // there shouldn't be any value initially  
   expect(fixture.debugElement.nativeElement.innerHTML).toBe('');  
     
   // let's set the @Input() value of our age property  
   component.age = '45';  
   // call detectChanges() so the change to the input is picked up  
   fixture.detectChanges();  
     
   // test to confirm that the element has the same value  
   expect(fixture.debugElement.nativeElement.innerHTML).toBe('45');  
});

This example first checks that there isn’t a value for the element that displays the passed-in age property. Then we set the @Input() value of age to be **45**. This is how we can set the properties of this @Input().

Next, we call detectChanges() in order for Angular to re-run the lifecycle hooks of the component so that the change to the age property is made. And, finally, we have the expectation expression that checks if the value displayed in the HTML element matches the value of the @Input() property.

### Tests for @Output() attribute

Testing an @Output() is slightly more complex than testing an @Input(). The reason for this is that an @Output() could emit an event that we need to check has been fired. Again, using our previous example, let’s have a test that checks that when a button in the template is clicked an event is emitted:

it('should test the @Output using a spy', () => {  
   // using a Spy to track the 'emit' event of the login EventEmitter  
   spyOn(component.login, 'emit');  
     
   // find the loginButton from the template  
   const loginButton = fixture.nativeElement.querySelector('button');  
     
   // trigger the click of the button  
   loginButton.click();  
     
   // test that the emit of the login EventEmitter has been called  
   expect(component.login.emit).toHaveBeenCalled();  
});

In this example, we’re using a spy from Jasmine to create a way to track @Output() of our component, which is a login event. As the @Output is using the EventEmitter class (we looked at the EventEmitter as part of the [Components, Templates, and Forms chapter](https://www.educative.io/collection/page/10370001/4603693004488704/6556515983949824)), it will have an ‘emit’ event, which we can track.

We find the loginButton again using the fixture wrapper to access the Template. Once we’ve found the button element, we trigger its click() event, which will fire off the login EventEmitter. Finally, in the expectation expression, we can check with the spy to see if the emit has been called. This means that our @Output() has successfully been fired.

With Angular using a component-based architecture, the @Input() and @Output() attributes can be used to pass data in and out of components. Being able to write tests that cover these interactions helps, as our application grows in complexity, to make sure our changes do not break this connection between the components. Understanding how to write tests that cover how data is passed between components is an extremely important part of writing Angular applications.

## Tests for a service

As well as being able to pass data via @Input() and @Output() attributes, another way data is passed throughout our applications is through the use of a service. So, we need to be able to write tests that will cover the functionality of a service.

### Example 1: Simple service

A test spec for a service is slightly different from a component test spec. If we look at the spec for our Demo Service, we can see that it has less setup than a component spec:

###### /

e2e

src

app

demo.service.spec.ts

demo.service.ts

app-routing.module.ts

app.component.html

app.component.sass

app.component.spec.ts

app.component.ts

app.module.ts

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.sass

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**demo.service.spec.ts**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

import { TestBed } from '@angular/core/testing';

import { DemoService } from './demo.service';

describe('DemoService', () => {

  let service: DemoService;

  beforeEach(() => {

    TestBed.configureTestingModule({});

    service = TestBed.inject(DemoService);

  });

  it('should be created', () => {

    expect(service).toBeTruthy();

  });

});





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/debug.html>

The auto-generated test spec file for service

As you can see, there is only one beforeEach() function, which is creating a TestingModule without any settings. Then the test is creating a local instance of the service we’re testing using the TestBed class and then testing that the service exists.

This is a fairly straightforward example. It shows how we again use the TestBed class for accessing classes we’re testing.

### Example 2: Complex service

Let’s have a look at another example of a more complex service:

import { HttpClientTestingModule, HttpTestingController } from  
   '@angular/common/http/testing';  
import { of } from 'rxjs/observable/of';  
import { TestBed } from '@angular/core/testing';  
import { BookService } from './book.service';  
     
describe('BookService', () => {  
    let serviceToTest: BookService;  
  
    beforeEach(() => {  
       TestBed.configureTestingModule({  
           imports: [HttpClientTestingModule],  
           providers: [BookService]  
       });  
       // inject the service using the TestBed  
       serviceToTest = TestBed.get(BookService);  
    });  
  
    it('should have a service instance', () => {  
       expect(serviceToTest).toBeDefined();  
    });  
         
    it('should return the mocked data in the subscribe', () => {  
        const spy = spyOn(serviceToTest,'getBookTitle').and.returnValue(  
            of({ title: 'Getting Started With Angular})  
        );  
             
        // subscribing to the method of the service  
        serviceToTest.getBookTitle().subscribe(result => {  
        // in the subcribe handler checking for the expected result  
        expect(result.title).toBe('Getting Started With Angular');  
       });  
        expect(spy).toHaveBeenCalled();  
   });  
         
    it('should not invoke the error throwing function since we mocked it', () => {  
        const mockFunction = () => {};  
        const spy = spyOn(serviceToTest, 'errorHandler').and.callFake(mockFunction);  
        serviceToTest.errorHandler();  
        expect(spy).toHaveBeenCalled();  
    });  
});

In this example, our BookService class has two functions, one that gets a book title and another that checks an error handler is working as expected. Again, you’ll see the use of the TestBed class to load in our service under test (BookService), which we set to a local property called serviceToTest.

#### Step 1: Create a spy

Then in our first test, we create a spy for the getBookTitle() method of our BookService. Not only are we creating a spy, but we’re also returning a value when the spy is created. In this instance, it’s an Observable created using the of() Operator that contains an object with a book title property.

📝 **Note:** See how we are able to use RxJs within our Jasmine based code. This is because Jasmine and RxJs are both using JavaScript, so there is no issue mixing the two different frameworks together. You can use other RxJs Operators within your tests. We learned about Operators in the [Observables and RxJs chapter](https://www.educative.io/collection/page/10370001/4603693004488704/4959879964917760).

#### Step 2: Call the getBookTitle() method

Once our spy has been established, we call the getBookTitle() method of the service. This runs the spy object, and as it returns an Observable, we need to subscribe to it. In the **subscribe** method, we’re using an assertion to test that the return result object has the value we’ve set. This confirms that the getBookTitle() method is working. After that, we have another assertion that confirms that our spy has been called. So, not only do we know that our service under test returns a value from the getBookTitle() method, but also that the getBookTitle() runs as expected.

#### Step 3: Checking error handler

The second test is checking that an errorHandler() method of our service is working. In this example, we’re first creating a mock function, which returns an empty object. The callFake() method of Jasmine means that all calls to this spy will return this fake function. In this example, it will return mockFunction(). Using callFake() is extremely useful because we can create different spies, all returning different fake functions so that we can test different scenarios in our tests. For example, in the mockFunction() we’re returning an empty object. We could have a second spy that returns an error message like this:

const mockFunctionWithMessage = () => {   
    return 'An error has occurred'   
}  
  
const spyWithMessage = spyOn(serviceToTest, 'errorHandlerWithMessage').and.callFake(mockFunctionWithMessage);

A new spy object returning something different. Spies are very useful for testing service functions and different responses from these service functions.

The official Angular documentation has some great examples of other types of tests you may write, not only for services but components as well. You can find these examples [here](https://angular.io/guide/testing#testing).

# Building a Release Version with the CLI

Let's explore how we can build a release version of an application with the Angular CLI.

**We'll cover the following**

* [ng build command](https://www.educative.io/courses/getting-started-with-angular/gx1P8xDz4vl#ng-build-command)
* [ng build --watch command](https://www.educative.io/courses/getting-started-with-angular/gx1P8xDz4vl#ng-build---watch-command)

## ng build command

The first thing we’re going to be looking at is how we can use the Angular CLI (hopefully, by now, you know how helpful the Angular CLI is) to create a production build.

So far, we’ve seen commands such as ng serve and ng test that we can use to either run the application locally in the browser or run through all our unit tests. There is another command we can run in the CLI that will create a release version of the application. The new command is as follows:

ng build

This will take our application code and make a release build in a new folder, which can be found in the dist folder in the project tree.

📝 **Note:** Click the terminal window and see what happens!

**Terminal 1**

Terminal

**Click to Connect...**

****

Now the CLI won’t show much in the Terminal once this has finished running, but if we look in the project directory, we can see a new dist folder as shown below:

📝 **Note:** We just want to show you the dist folder so the following SPA widget won’t run.

###### /

dist

Client-Contacts-Manager-Angular

favicon.ico

index.html

main.js

main.js.map

polyfills.js

polyfills.js.map

runtime.js

runtime.js.map

styles.js

styles.js.map

vendor.js

vendor.js.map

e2e

src

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**main.js**

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

84

85

86

87

/\* harmony import \*/ var \_angular\_router\_\_WEBPACK\_IMPORTED\_MODULE\_2\_\_ = \_\_webpack\_require\_\_(/\*! @angular/router \*/ "tyNb");

function AppComponent\_pre\_85\_Template(rf, ctx) { if (rf & 1) {

    \_angular\_core\_\_WEBPACK\_IMPORTED\_MODULE\_0\_\_["ÉµÉµelementStart"](0, "pre");

    \_angular\_core\_\_WEBPACK\_IMPORTED\_MODULE\_0\_\_["ÉµÉµtext"](1, "ng generate component xyz");

    \_angular\_core\_\_WEBPACK\_IMPORTED\_MODULE\_0\_\_["ÉµÉµelementEnd"]();

} }

function AppComponent\_pre\_86\_Template(rf, ctx) { if (rf & 1) {

    \_angular\_core\_\_WEBPACK\_IMPORTED\_MODULE\_0\_\_["ÉµÉµelementStart"](0, "pre");

    \_angular\_core\_\_WEBPACK\_IMPORTED\_MODULE\_0\_\_["ÉµÉµtext"](1, "ng add @angular/material");

    \_angular\_core\_\_WEBPACK\_IMPORTED\_MODULE\_0\_\_["ÉµÉµelementEnd"]();

} }

function AppComponent\_pre\_87\_Template(rf, ctx) { if (rf & 1) {

    \_angular\_core\_\_WEBPACK\_IMPORTED\_MODULE\_0\_\_["ÉµÉµelementStart"](0, "pre");

    \_angular\_core\_\_WEBPACK\_IMPORTED\_MODULE\_0\_\_["ÉµÉµtext"](1, "ng add @angular/pwa");

    \_angular\_core\_\_WEBPACK\_IMPORTED\_MODULE\_0\_\_["ÉµÉµelementEnd"]();

} }

function AppComponent\_pre\_88\_Template(rf, ctx) { if (rf & 1) {

    \_angular\_core\_\_WEBPACK\_IMPORTED\_MODULE\_0\_\_["ÉµÉµelementStart"](0, "pre");

    \_angular\_core\_\_WEBPACK\_IMPORTED\_MODULE\_0\_\_["ÉµÉµtext"](1, "ng add \_\_\_\_\_");

    \_angular\_core\_\_WEBPACK\_IMPORTED\_MODULE\_0\_\_["ÉµÉµelementEnd"]();

} }

function AppComponent\_pre\_89\_Template(rf, ctx) { if (rf & 1) {

    \_angular\_core\_\_WEBPACK\_IMPORTED\_MODULE\_0\_\_["ÉµÉµelementStart"](0, "pre");

    \_angular\_core\_\_WEBPACK\_IMPORTED\_MODULE\_0\_\_["ÉµÉµtext"](1, "ng test");

    \_angular\_core\_\_WEBPACK\_IMPORTED\_MODULE\_0\_\_["ÉµÉµelementEnd"]();

} }

function AppComponent\_pre\_90\_Template(rf, ctx) { if (rf & 1) {

    \_angular\_core\_\_WEBPACK\_IMPORTED\_MODULE\_0\_\_["ÉµÉµelementStart"](0, "pre");

    \_angular\_core\_\_WEBPACK\_IMPORTED\_MODULE\_0\_\_["ÉµÉµtext"](1, "ng build --prod");





Run

6441677 character(s) over limit (Max: 700KB)

Save

Reset

**Your app can be found at:**[https://ed-6434592358924288.educative.run](https://ed-6434592358924288.educative.run/)

dist folder containing all the minified code

You can see the new dist folder, which has a sub-folder that uses the name of our project as its name. In this sub-folder is all the minified code the build process has made. Notice that there are no TypeScript files here. All our sub-folders have gone, and all of our **component** and **service** folders are no longer there. This is because the build process has compiled all our code into one single JavaScript file, the main.js file.

📝 **Note:** If you open the main.js file and scroll through it, you’ll see how the TypeScript compiler has converted our TypeScript into JavaScript. You’ll also notice some of the TypeScript we’ve written, but now it’s in JavaScript.

Packaging Angular application

## ng build --watch command

The ng build command will generate this distribution folder, but like so many other Angular CLI commands, we can pass an argument along with this build command. If we use ng build --watch, this will tell the CLI to create a dist folder and then watch for any further changes to our code.

If we make any changes to the codebase, the CLI will automatically generate a new version of the dist folder by first clearing out what is already there and then generating a new version from the latest code. This is extremely helpful when you’re getting a final release ready, and you have to make those final, small tweaks for the release.

# The Environment Files

Learn about standard environment files within our application as well as other environment files we can create for our application.

**We'll cover the following**

* [The environment settings files](https://www.educative.io/courses/getting-started-with-angular/RMyWorp1ZxE#The-environment-settings-files)
  + [Example](https://www.educative.io/courses/getting-started-with-angular/RMyWorp1ZxE#Example)
* [Creating alternative environment files](https://www.educative.io/courses/getting-started-with-angular/RMyWorp1ZxE#Creating-alternative-environment-files)
  + [Create QA version of the environments files](https://www.educative.io/courses/getting-started-with-angular/RMyWorp1ZxE#Create-QA-version-of-the-environments-files)
  + [Add details of environment to angular.json file](https://www.educative.io/courses/getting-started-with-angular/RMyWorp1ZxE#Add-details-of-environment-to-angular.json-file)
* [CLI commands for building different environments](https://www.educative.io/courses/getting-started-with-angular/RMyWorp1ZxE#CLI-commands-for-building-different-environments)
  + [Standard build](https://www.educative.io/courses/getting-started-with-angular/RMyWorp1ZxE#Standard-build)
  + [Production build](https://www.educative.io/courses/getting-started-with-angular/RMyWorp1ZxE#Production-build)
  + [Build using a configuration](https://www.educative.io/courses/getting-started-with-angular/RMyWorp1ZxE#Build-using-a-configuration)

## The environment settings files

Within our application, we have two environment files as standard. There’s one for development,environment.ts, and one for production, environment.prod.ts. Both of these files can be found in the /environments folder of our application. Within these files, we can add any environmental settings we want to access within our code.

### Example

For example, we could set the main API URL. Then, in the development version, we have a development URL, which points to a dev server. And in the production environments file, the same URL points to the live product URL.

This is what the environment.ts file could look like with an API URL for development:

export const environment = {  
    // set to false to show that not a production version  
    production: false,   
         
    baseApiURL: 'https://dev.api.example.com'  
};

In the following code, we have the same properties, but this time in the environment.prod.ts file:

export const environment = {  
   production: true,  
   baseApiURL: 'https://api.example.com'  
};

Within our code, we can access these environmental properties by importing the environment file, which gives us access to the properties within the file:

import { environment } from 'src/environments/environment';  
...  
ngOnInit() {  
   console.log(environment.baseApiURL);  
}  
...

## Creating alternative environment files

While, by default, Angular provides us with two environment files (development and production), we can actually create other environment files if we need to have a different set of values for our environment properties on another server. For example, we may have a situation where the project we’re working on has a **QA server**. So, our baseApiURL property for the QA server might be different, and we need to have an environment file for that QA server.

### Create QA version of the environments files

In order to add a new environment.ts file, we need to make two changes. First, we need to create our new QA version of the environment files. To do this, we can simply create an environment.qa.ts file in the /environments folder. This QA version needs to have all the same properties as the other two files, but with values that work on our QA server.

### Add details of environment to angular.json file

The next step is to add the details of this new environment to the angular.json file. Within this file, there is a section called **configurations** (line 35 of the JSON file). This is where we can add alternative configuration settings:

`

###### /

e2e

src

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**angular.json**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

{

  "$schema": "./node\_modules/@angular/cli/lib/config/schema.json",

  "version": 1,

  "newProjectRoot": "projects",

  "projects": {

    "Client-Contacts-Manager-Angular": {

      "projectType": "application",

      "schematics": {

        "@schematics/angular:component": {

          "style": "scss"

        }

      },

      "root": "",

      "sourceRoot": "src",

      "prefix": "app",

      "architect": {

        "build": {

          "builder": "@angular-devkit/build-angular:browser",

          "options": {

            "outputPath": "dist/Client-Contacts-Manager-Angular",

            "index": "src/index.html",

            "main": "src/main.ts",

            "polyfills": "src/polyfills.ts",

            "tsConfig": "tsconfig.app.json",

            "aot": true,

            "assets": [

              "src/favicon.ico",

              "src/assets"

            ],

            "styles": [

              "src/styles.scss"

            ],





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/debug.html>

configurations section in angular.json file

Note that in this extract from the angular.json file, we have just one configuration for production. In order to add the new QA settings, we just add a new entry within this configurations section:

"configurations": { "production": {  
    "fileReplacements": [  
       {  
           "replace": "src/environments/environment.ts",  
           "with": "src/environments/environment.prod.ts"  
       }],  
        ...   
    },  
    "qa": { "fileReplacements": [{  
       "replace": "src/environments/environment.ts",  
       "with": "src/environments/environment.qa.ts"  
        } ],  
        ...   
    },  
}

Now, we have a QA entry, as well as the production entry. The thing to notice here is the fileReplacements section. This is where we’re telling the Angular CLI when we are running in production in order to replace the development version of the environment.ts file with the environment.prod.ts file. With the QA version, we’re telling the CLI to replace the environment.ts file with environment.qa.ts. Then, when we access an environmental property in either of these configurations, the correct settings are used.

## CLI commands for building different environments

With this new configuration set up, we can use the CLI to build our application in either of these configurations. When we call ng build, we add extra commands to tell the CLI to build for the different environments.

### Standard build

For example, to perform a standard build, we can use the following code:

ng build

### Production build

To tell the CLI to build for production, we can run the following command:

ng build --prod=true

This will set the configuration to use the production settings (our environment.prod.ts file will be used), which makes use of bundling (creating those JavaScript files we saw in the dist folder) and uses tree-shaking to remove any unused code, reducing the size of the final output.

### Build using a configuration

We can also tell the CLI to use our new configuration. This is helpful if we want to test that the configuration works before committing this change to our main code branch:

ng build --configuration=qa

This command will access the QA section of the configuration portion from the angular.json file.

If we want to actually serve the application to a local browser using this configuration, we can do so by using the ng serve command:

ng serve --configuration=qa

Alternatively, we can use the following command:

ng serve --configuration=prod

Again, by showing you how powerful and helpful the Angular CLI is, we can set up multiple configurations for however many environments we may have and both test them and build them before committing to the main codebase.

# The angular.json File

Let's learn about the different sections of the angular.json file.

**We'll cover the following**

* [Angular CLI workspace file](https://www.educative.io/courses/getting-started-with-angular/3joJOBjKn3p#Angular-CLI-workspace-file)
  + [The main source of configuration for the entire project](https://www.educative.io/courses/getting-started-with-angular/3joJOBjKn3p#The-main-source-of-configuration-for-the-entire-project)
* [The schema section](https://www.educative.io/courses/getting-started-with-angular/3joJOBjKn3p#The-schema-section)
* [The Projects section](https://www.educative.io/courses/getting-started-with-angular/3joJOBjKn3p#The-Projects-section)
* [The Architect section](https://www.educative.io/courses/getting-started-with-angular/3joJOBjKn3p#The-Architect-section)
  + [The build section](https://www.educative.io/courses/getting-started-with-angular/3joJOBjKn3p#The-build-section)
  + [The serve section](https://www.educative.io/courses/getting-started-with-angular/3joJOBjKn3p#The-serve-section)
  + [The extract-i18n section](https://www.educative.io/courses/getting-started-with-angular/3joJOBjKn3p#The-extract-i18n-section)
  + [The test section](https://www.educative.io/courses/getting-started-with-angular/3joJOBjKn3p#The-test-section)
  + [The lint section](https://www.educative.io/courses/getting-started-with-angular/3joJOBjKn3p#The-lint-section)
* [The tslint.json file](https://www.educative.io/courses/getting-started-with-angular/3joJOBjKn3p#The-tslint.json-file)
  + [Example](https://www.educative.io/courses/getting-started-with-angular/3joJOBjKn3p#Example)

## Angular CLI workspace file

In the previous section, we mentioned the angular.json file. This is an extremely important part of an Angular application, so it is worth exploring this file some more.

The angular.json file is always found in the root folder of a CLI generated project, along with the package.json file. Originally, this file was named .angular-CLI.json, but as of Angular 6+, the file was renamed to angular.json, though its official name is the **Angular CLI workspace file**.

### The main source of configuration for the entire project

This file is used as the main source of configuration for the entire project, as we’ve already seen in the configuration section and the environment settings file. There are many more settings that we can change within this JSON file in order to optimize our application. Let’s have a look at some of the sections of this file.

###### /

e2e

src

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**angular.json**

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

                {

                  "replace": "src/environments/environment.ts",

                  "with": "src/environments/environment.prod.ts"

                }

              ],

              "optimization": true,

              "outputHashing": "all",

              "sourceMap": false,

              "extractCss": true,

              "namedChunks": false,

              "extractLicenses": true,

              "vendorChunk": false,

              "buildOptimizer": true,

              "budgets": [

                {

                  "type": "initial",

                  "maximumWarning": "2mb",

                  "maximumError": "5mb"

                },

                {

                  "type": "anyComponentStyle",

                  "maximumWarning": "6kb",

                  "maximumError": "10kb"

                }

              ]

            }

          }

        },

        // serve section

        "serve": {

          "builder": "@angular-devkit/build-angular:dev-server",

          "options": {

            "browserTarget": "Client-Contacts-Manager-Angular:build"





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/debug.html>

angular.json file

## The schema section

The first part of the JSON file defines the schema that this file should be checked against and the version of the application. After this, we have the projects section.

"$schema": "./node\_modules/@angular/cli/lib/config/schema.json",  
  "version": 1,

## The Projects section

In this section of JSON, the projects under the CLI workspace are listed. Currently, we just have the main Angular project, but if we have a far more complex application that is made up of several projects, they are listed here, along with the settings for each project.

We could have a large enterprise application that is not only made up of the core Angular project but is also making use of multiple library projects. These library projects could contain reusable code that is being used across multiple Angular projects. In the projects section, we can define the CLI settings for each library project, along with the settings of that main project.

📝 **Note:** If you are planning a large-scale Angular application and you want to make use of library projects so that you can reuse code in your application, then Nrwl’s NX solution may help. It extends the Angular CLI, so these library projects can be set up and managed through the [NX Extensions](https://nx.dev/).

Within the Projects section, we have a single sub-section called the **architect section**. Not only do we have the main Angular project, but the settings for the end-to-end tests version of the application are kept in this Projects section. This project contains the settings that CLI needs in order to run the end-to-end tests of the application.

So, if we have a complex application made up of the main core application, a set of the end-to-end tests, and a library project, we would have three sub-sections within the Projects section:

* Client-contacts-manager-angular: The main application
* Client-contacts-manager-angular-e2e: The end-to-end tests settings
* CCM-Useable-code-library: A reusable code library

## The Architect section

In this section, we have the settings for the various ways the application is going to be built by the CLI. For the main Angular application, we have five sub-parts in this section as follows:

* build
* serve
* extract-i18n
* test
* lint

### The build section

The build section contains all the options for how the application will be built by the CLI and where the CLI can find various files it needs when building the application. This includes the configuration settings we looked at earlier, as well as any third-party libraries we need to add to our application. These libraries can be listed in the scripts array of the options section.

### The serve section

The serve section has the necessary settings so that the CLI knows what build scripts it should use. These scripts come as part of Angular and are defined in the main package.json file, under the scripts array:

"scripts": {  
   "ng": "ng",  
   "start": "ng serve",  
   "build": "ng build",  
   "test": "ng test",  
   "lint": "ng lint",  
   "e2e": "ng e2e"  
},

We can also add our own build scripts to this section of package.json if we need to have a specific build process for our environments.

### The extract-i18n section

Next is the extract-i18n section. This tells the CLI to extract all the i18n messages from our application. This is the detail that is needed so that we can add internationalization (multi-language support) to our application.

📝 **Note:** For more information on i18n in Angular, check out the [official documentation](https://angular.io/guide/i18n).

### The test section

Next, the test section is very similar to the serve section, but it is using Karma to build and run the application.

### The lint section

Last but not least is the lint section. This tells the CLI to check our TypeScript files against the Lint settings that come as part of the application. These lint settings check that our TypeScript code is following the rules set out in the tslint.json file.

## The tslint.json file

**Linting** or **linters** will check our code for any potential problems. It will highlight what these problems are and possible ways to solve these issues. When working on large projects with multiple team members, we can define a set of rules for how we want the code of the application to be written.

### Example

For example, if we want to use double quotes instead of single quotes, or tabs instead of spaces, these rules can be added to the tslint.json file. Then, when the application is saved or when we run ng lint, these rules are checked against our code.

By default, there are a large set of rules already in the tslint.json file, but if you want to add more settings, you can. For a complete list of the rules that are available, check out the [official TSLint rules page](https://palantir.github.io/tslint/rules/).

# Ahead-of-time Compilation

Let's explore how we can create an optimized, production-ready version of our application using the Ahead-of-Time compilation.

**We'll cover the following**

* [What is an Ahead-of-Time compilation?](https://www.educative.io/courses/getting-started-with-angular/mEnQljNQlNA#What-is-an-Ahead-of-Time-compilation?)
  + [Advantage](https://www.educative.io/courses/getting-started-with-angular/mEnQljNQlNA#Advantage)
  + [Example](https://www.educative.io/courses/getting-started-with-angular/mEnQljNQlNA#Example)
* [JIT versus AoT](https://www.educative.io/courses/getting-started-with-angular/mEnQljNQlNA#JIT-versus-AoT)
  + [Serve application with AoT](https://www.educative.io/courses/getting-started-with-angular/mEnQljNQlNA#Serve-application--with-AoT)
  + [Build an application with AoT](https://www.educative.io/courses/getting-started-with-angular/mEnQljNQlNA#Build-an-application-with-AoT)
* [AoT in production](https://www.educative.io/courses/getting-started-with-angular/mEnQljNQlNA#AoT-in-production)
* [Why we should use AoT](https://www.educative.io/courses/getting-started-with-angular/mEnQljNQlNA#Why-we-should-use-AoT)
  + [Fewer asynchronous calls](https://www.educative.io/courses/getting-started-with-angular/mEnQljNQlNA#Fewer-asynchronous-calls)
  + [Smaller Angular framework download](https://www.educative.io/courses/getting-started-with-angular/mEnQljNQlNA#Smaller-Angular-framework-download)
  + [Detect template errors earlier](https://www.educative.io/courses/getting-started-with-angular/mEnQljNQlNA#Detect-template-errors-earlier)
  + [Better security](https://www.educative.io/courses/getting-started-with-angular/mEnQljNQlNA#Better-security)

## What is an Ahead-of-Time compilation?

**Ahead-of-Time** compilation, or **AoT**, as it is more commonly called, is the ability that Angular has where it can convert our HTML and TypeScript into JavaScript before the browser downloads and uses this JavaScript.

This conversion is performed by the AoT compiler, which compiles our HTML and TypeScript into JavaScript files just before they are needed.

### Advantage

The benefit this brings is that when a user is going to access an application in the browser, they don’t have to wait for the compiler to compile all the HTML and TypeScript of the application into JavaScript.

### Example

If we have an application that has over 100 components and templates plus 50 services, this is a large amount of HTML and TypeScript that needs to be compiled before the JavaScript is ready to be downloaded by the users’ browser. This delays our application from being usable, making it appear as though the application is slow to start, which is not good for a modern web application.

## JIT versus AoT

In Angular, there are two ways that code can be delivered, either using **Just-in-Time** (JIT) compilation or AoT.

**Just-in-Time compilation** is where our code is compiled within the browser at runtime.

This is commonly used when we run ng serve, and ng build (not ng build --prod – this switches to using AoT).

AoT will compile the code before it is downloaded to the browser, removing this slight lag we see when the browser is compiling the code at runtime. The reason that ng serve and ng build aren’t using AoT by default is that when we’re running ng serve, we’re running the application locally for development purposes, and this lag is not an issue for us like it would be with end-users.

### Serve application with AoT

We can tell both the ng serve and ng build commands to switch to using AoT if we want to serve the application with AoT switched on to check how it performs. This is done by passing in an argument to these two commands.

ng serve --aot

### Build an application with AoT

Now, if we want to build the application so that the release version uses the Ahead-of-Time compilation, we would use the following command:

ng build --prod

This makes a build version that uses AoT instead of the JIT compilation.

📝 **Note:** It’s important to remember that we use JIT compilation locally when developing our application, whereas AoT is for production. The reasons are that the code that’s generated through this --aot command is **more secure** (it’s harder to read in the browser, unlike development code).

Also, when you’re using AoT in the build version, the Angular compiler is not part of the release package, thus making it as small as possible.

## AoT in production

When we were looking at the angular.json file, you may have noticed in the Production sub-section of the configuration section that there is a setting for AoT:

"configurations": {  
   "production": {  
       "fileReplacements": [  
           {  
               "replace": "src/environments/environment.ts",  
               "with": "src/environments/environment.prod.ts"  
           }],  
        "optimization": true,   
        "outputHashing": "all",   
        "sourceMap": false,   
        "extractCss": true,   
        "namedChunks": false,   
        "aot": true,  
        ....

Notice that the aot property is set to **true**. This means that the Ahead-of-Time compilation is on by default in production. Therefore, we don’t need to worry if it’s on or not when releasing our application to production. It’s already set for us.

## Why we should use AoT

As we briefly mentioned earlier, the application is usually compiled, and all our HTML and TypeScript is compiled to JavaScript files. If this compilation is performed on the browser, then this can take a short while, depending on the size of our application. Since internet connections have gotten faster and faster over the last few years, users expect to see websites and web apps load almost instantly. Studies have been conducted and shown that if a website doesn’t load within a few seconds, the user will leave the site.

So, with the time an application loads being a big factor on whether a user stays on an application, being able to show the application working as soon as possible is extremely important. Through AoT, we remove this browser compilation phase, making the app appear in the browser almost immediately.

📝 **Note:** Remember that TypeScript is a superset of JavaScript and cannot actually be run in the browser. This is why we need to have TypeScript compiled down to JavaScript, which the browser understands.

You can test this by running your Angular applications using ng serve and ng serve --aot.

However, that’s not all. There are a number of other benefits AoT brings.

### Fewer asynchronous calls

All the CSS and HTML files are combined, reducing the number of calls the application needs to make to the server to access separate CSS and HTML files.

### Smaller Angular framework download

With the application already compiled, the Angular compiler doesn’t need to be downloaded to the browser. This reduces the size of the Angular framework by removing the Angular compiler.

### Detect template errors earlier

Errors in our HTML templates can be found during this build step. To ensure that the end-user doesn’t see an error in the template, we pick these errors up earlier.

### Better security

With all the code compiled to JavaScript before getting to the browser, there are no HTML templates for injection attacks to be made through.

Ahead-of-Time compilation is a great feature of Angular and brings so many benefits. This is why it is switched on by default in the production version of the application.

# Other Various Production Optimisations

Let's explore how we can optimize our application for production by setting the browser support level.

**We'll cover the following**

* [Setting the browser support level](https://www.educative.io/courses/getting-started-with-angular/7XZVLNROExj#Setting-the-browser-support-level)
* [Browser-specific CSS](https://www.educative.io/courses/getting-started-with-angular/7XZVLNROExj#Browser-specific-CSS)
  + [What is Autoprefixer?](https://www.educative.io/courses/getting-started-with-angular/7XZVLNROExj#What-is-Autoprefixer?)
  + [Create browser-specific attributes](https://www.educative.io/courses/getting-started-with-angular/7XZVLNROExj#Create-browser-specific-attributes)
  + [Add support for browsers in Angular application](https://www.educative.io/courses/getting-started-with-angular/7XZVLNROExj#Add-support-for-browsers-in-Angular-application)
* [Enabling Production mode](https://www.educative.io/courses/getting-started-with-angular/7XZVLNROExj#Enabling-Production-mode)
* [What is Production mode?](https://www.educative.io/courses/getting-started-with-angular/7XZVLNROExj#What-is-Production-mode?)

## Setting the browser support level

As well as AoT and environmental settings, there are other options we can set in order to optimize our application for production. One of these areas is setting the browser support level you want for your application.

Setting browser level support

## Browser-specific CSS

If we’re developing an internal application that is solely being used within a corporate environment and in that environment they only allow the users access to one browser, we can optimize our application to work for that browser. Again, by using the CLI, we can add a setting to the angular.json file that tells the CLI to optimize the build for this type of browser.

### What is Autoprefixer?

The Angular CLI uses Autoprefixer, which is a plugin for CSS that will automatically prefix our CSS classes with the appropriate prefix.

### Create browser-specific attributes

If we have a class called panel and this class has some browser-specific attributes, we can create the CSS like this:

::panel {  
    color: blue;  
}

What the Autoprefixer will do is replace the :: with a browser-specific prefix. The panel class will now look as follows:

::-webkit-panel {  
       color: blue;  
}  
::-moz-panel {  
   color: blue;  
}  
::panel {  
   color: blue;  
}

The Autoprefixer plugin uses a library called Browserslist to get a list of the browsers it should add prefixing to (for example, we could exclude Safari). This Browserslist library uses a config setting called browserlist to set the list of browsers it needs to support. This config setting can either be set in the main package.json file or in a .browserlistrc file. The Angular CLI will then read the list from this file and add the appropriate CSS prefixes to the CSS we defined in our application.

###### /

e2e

src

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**.browserslistrc**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

# This file is used by the build system to adjust CSS and JS output to support the specified browsers below.

# For additional information regarding the format and rule options, please see:

# https://github.com/browserslist/browserslist#queries

# For the full list of supported browsers by the Angular framework, please see:

# https://angular.io/guide/browser-support

# You can see what browsers were selected by your queries by running:

#   npx browserslist

last 1 Chrome version

last 1 Firefox version

last 2 Edge major versions

last 2 Safari major versions

last 2 iOS major versions

Firefox ESR

not IE 9-10 # Angular support for IE 9-10 has been deprecated and will be removed as of Angular v11. To opt-in, remove the 'not' prefix on this line.

not IE 11 # Angular supports IE 11 only as an opt-in. To opt-in, remove the 'not' prefix on this line.





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/debug.html>

.browserlistrc file

### Add support for browsers in Angular application

Going back to our example, we have an application that shouldn’t support Internet Explorer (IE). We can add the following to the browserlist section of these settings:

"browserslist": [  
   "last 2 versions",  
   "not ie <= 10"  
]

In this example, we’re telling Browserslist to support only the last 2 versions of the browser the application is running in (this could be Chrome, Firefox, or Safari) and not to run in IE.

You may be wondering why this doesn’t actually list the specific browsers to run in. Well, the recommended approach from Browserslist is to only set the number of versions that should be supported, but if your application is to run within an organization that only allows one browser to be installed on users computers, then Browserslist will only allow support for the last two versions of that browser.

It is rare that the CSS settings are optimized like this because, with some applications, we can’t tie down the list of browsers it should support. However, with corporate applications that target a certain browser, we can make great use of this optimization feature.

## Enabling Production mode

We’ve already seen some of the build optimizations that are available to us in the angular.json file, but we can also tell Angular to run in Production mode instead of Development mode, which is the default mode. When we run ng serve, the application is being run in Development mode.

To switch to Production mode, all we need to do is add --prod as an argument to the command we’re running. The following list of commands are all in Production mode:

* ng serve --prod
* ng build --prod
* ng test --prod
* ng e2e --prod

To check whether a command can be run in Production mode, check the list of CLI commands from the [Angular documentation](https://cli.angular.io/).

## What is Production mode?

When any of the preceding commands are running in Production mode, it means that Angular is running the application faster than Development mode. The Angular compiler manages to do this by switching off some settings, including the following:

* Dual change detection cycles
* Stops outputting important warnings, but errors are still displayed
* Stops building a debugging elements tree

These types of settings may speed up the application, but for debugging purposes, they are extremely helpful. This is why it’s good to have Development mode enabled when developing and debugging an application. When running the application in production, these settings can be disabled, which is what they’re automatically set to.

# Making Use of Lazy Loading

Let's explore how we can make use of lazy loading for optimizing our application for production.

**We'll cover the following**

* [Implement lazy loading](https://www.educative.io/courses/getting-started-with-angular/m7D7w0X4wR3#Implement-lazy-loading)
  + [app-routing.module.ts file](https://www.educative.io/courses/getting-started-with-angular/m7D7w0X4wR3#app-routing.module.ts-file)
  + [client.module.ts file](https://www.educative.io/courses/getting-started-with-angular/m7D7w0X4wR3#client.module.ts-file)

Another way performance can be improved is through the use of **lazy loading** within our application. Back in [NgModules chapter](https://www.educative.io/collection/page/10370001/4603693004488704/4518520179130368" \t "_blank), we introduced you to the concept of modules in Angular. This is how our application is structured using a module. This contains all the Components, Services, Templates, and other files that are needed for a section.

## Implement lazy loading

Through modules, we can create **feature modules**. These are modules that contain all the functionality of a section of our application (think of ClientModule or CompanyModule of our demo application). Through the use of modules, we can use lazy loading to only load the modules (and the code contained within that module) when needed.

### app-routing.module.ts file

Lazy loading is configured through the Routes of our application. Usually, we would load the main component for a route like this:

{  
   path: 'clients/new',  
   component: ClientPageComponent  
},

Instead, we set up the Route so that it loads the module that the Component belongs to, like this:

{  
    path: 'clients/new',  
    loadChildren: './client/client.module#ClientModule'  
},

### client.module.ts file

Then, within the ClientModule, we have a local route to load the individual Routes within that module. So if we need to client/new or client/search, these two Routes are defined in the ClientModule itself, like this:

const routes: Routes = [  
   {  
       path: 'new',  
       component: ClientPageComponent  
   },  
   {  
       path: 'search',  
       component: ClientSearchPageComponent  
   }];

📝 **Note:** Press the **RUN** button to compile and serve the application. Once the app compiles, click on the URL given after Your app can be found at, and see what happens!

###### /

e2e

src

app

about

clients

client-detail

client-details-page

client-edit-page

client-form

client-list

client-page

client-search-page

client.module.ts

client.service.spec.ts

client.service.ts

client.ts

company

navigation

page-not-found

search

services

shared

app-routing.module.ts

app.component.html

app.component.scss

app.component.spec.ts

app.component.ts

app.module.ts

custom-material.module.ts

assets

environments

favicon.ico

index.html

main.ts

polyfills.ts

styles.scss

test.ts

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**client.module.ts**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

import { CommonModule } from '@angular/common';

import { CUSTOM\_ELEMENTS\_SCHEMA, NgModule } from '@angular/core';

import { FormsModule, ReactiveFormsModule } from '@angular/forms';

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

import { CustomMaterialModule } from '../custom-material.module';

import { SharedModule } from '../shared/shared.module';

import { ClientDetailComponent } from './client-detail/client-detail.component';

import { ClientDetailsPageComponent } from './client-details-page/client-details-page.component';

import { ClientEditPageComponent } from './client-edit-page/client-edit-page.component';

import { ClientFormComponent } from './client-form/client-form.component';

import { ClientItemComponent } from './client-list/client-item/client-item.component';

import { ClientListComponent } from './client-list/client-list.component';

import { ClientPageComponent } from './client-page/client-page.component';

import { ClientSearchPageComponent } from './client-search-page/client-search-page.component';

import { ClientService } from './client.service';

const routes: Routes = [

  {

    path: 'clients/new',

    component: ClientPageComponent

  },

  {

    path: 'clients/search',

    component: ClientSearchPageComponent

  },

  {

    path: 'clients/details/:id',

    component: ClientDetailsPageComponent

  },

  {

    path: 'clients/edit/:id',

    component: ClientEditPageComponent





Run

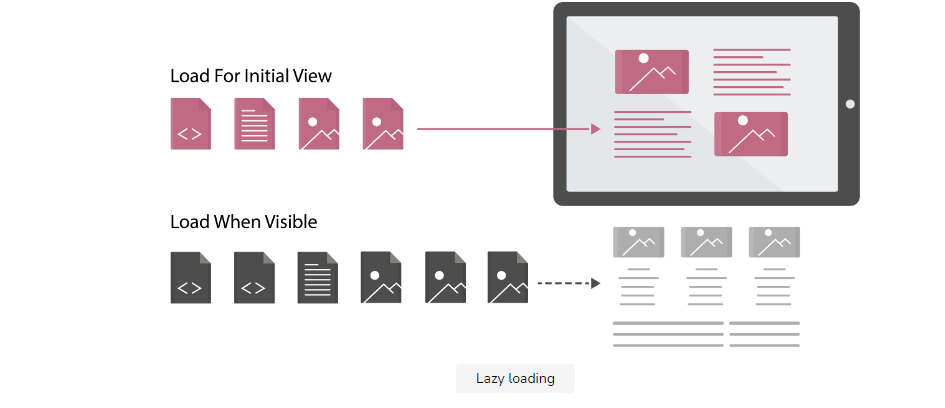
Save

Reset

**Your app can be found at:**[https://ed-6434592358924288.educative.run](https://ed-6434592358924288.educative.run/)

Client Contacts Manager Application with lazy loading

Through using this lazy loading approach of setting up Routes, when a Route is loaded, the module is loaded along with all its children, and then the local route searches for the matching component. Any other Routes not being called at that time don’t load, along with any of their children’s components, and this reduces the size of the download in the browser. Lazy loading, along with Ahead-of-Time compilation, makes Angular load extremely fast.



# Application Size Budget

Let's explore how we can set an application size budget in the angular.json file.

**We'll cover the following**

* [Setting an application size budget](https://www.educative.io/courses/getting-started-with-angular/g7LkqoJmmYD#Setting-an-application-size-budget)
  + [Settings for application size](https://www.educative.io/courses/getting-started-with-angular/g7LkqoJmmYD#Settings-for-application-size)
  + [Types of size budgets](https://www.educative.io/courses/getting-started-with-angular/g7LkqoJmmYD#Types-of-size-budgets)

## Setting an application size budget

Another option for reducing the size of the download the browser needs to make, is to set an application size budget. This is where we can configure a size for the application that we think is the maximum, and during the build process, if our application has grown close to this maximum size, the Angular CLI will warn us.

During the development of an application, we add more and more packages and code as the app grows. This all leads to a final compiled file size that the browser needs to download. The larger the file, the longer the browser will take to download, and the longer it takes for our application to be ready to start (this is especially an issue on mobile devices).

In the angular.json file, under the configuration section, we can set an array of budgets (line 55):

###### /

e2e

src

.browserslistrc

.editorconfig

.gitignore

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**angular.json**

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

          // Architect section

          // build section

          "configurations": {

            "production": {

              "fileReplacements": [

                {

                  "replace": "src/environments/environment.ts",

                  "with": "src/environments/environment.prod.ts"

                }

              ],

              "optimization": true,

              "outputHashing": "all",

              "sourceMap": false,

              "extractCss": true,

              "namedChunks": false,

              "extractLicenses": true,

              "vendorChunk": false,

              "buildOptimizer": true,

              "budgets": [

                {

                  "type": "initial",

                  "maximumWarning": "2mb",

                  "maximumError": "5mb"

                },

                {

                  "type": "anyComponentStyle",

                  "maximumWarning": "6kb",

                  "maximumError": "10kb"

                }

              ]

            }

          }





Run

Save

Reset

**Your app can be found at:**<https://ed-6434592358924288.educative.run/debug.html>

angular.json file

### Settings for application size

There are a number of settings we can make for this section:

| **Settings** | **Purpose** |
| --- | --- |
| **type** | The type of budget – we can have more than one name: a name for the bundle budget type |
| **baseline** | A baseline size that’s used to compare against the final size of our generated application |
| **maximumWarning** | The max size for displaying a warning; uses the baseline to compare |
| **maximumError** | The max size for displaying an error; uses the baseline to compare |
| **minimumWarning** | The min size for displaying a warning |
| **minimumError** | The min threshold for displaying an error |
| **warning** | The threshold for displaying a warning relative to the baseline |
| **error** | The threshold for displaying an error relative to the baseline |

### Types of size budgets

One of the settings for the budget array is type. There are a few types of budgets that Angular supports, which are as follows:

| **Budget Type** | **Purpose** |
| --- | --- |
| **Bundle** | The size of the entire bundle |
| **Initial** | The size of the initial application itself |
| **AllScripts** | The size of all the scripts |
| **All** | The size of all the applications |
| **Any** | The size of any script |

With these different types, we can set different size budgets we want to check for. If at any point in the build the application is getting too big, the build process will warn us of these problems. Then, we can investigate to see what is causing the large file size and amend anything if needed.

Being able to keep an eye on the final size of the download the browser needs to make is extremely important in keeping a fast, optimized production application.

# Measuring Performance

Explore how we can measure the performance of our application using Chrome Dev Tools.

**We'll cover the following**

* [Chrome Dev Tools](https://www.educative.io/courses/getting-started-with-angular/mE2REOJ5R0R#Chrome-Dev-Tools)

## Chrome Dev Tools

After all these optimization settings, we need a way to measure our Angular application’s performance.

One way we can check the performance of the application is through the Chrome Dev Tools and, in particular, the Network Analysis section:

###### /

e2e

src

README.md

angular.json

karma.conf.js

package-lock.json

package.json

tsconfig.app.json

tsconfig.json

tsconfig.spec.json

tslint.json

**angular.json**

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

        }

      },

      "root": "",

      "sourceRoot": "src",

      "prefix": "app",

      "architect": {

        "build": {

          "builder": "@angular-devkit/build-angular:browser",

          "options": {

            "outputPath": "dist/Client-Contacts-Manager-Angular",

            "index": "src/index.html",

            "main": "src/main.ts",

            "polyfills": "src/polyfills.ts",

            "tsConfig": "tsconfig.app.json",

            "assets": [

              "src/favicon.ico",

              "src/assets"

            ],

            "styles": [

              "src/styles.scss"

            ],

            "scripts": []

          },

          "configurations": {

            "production": {

              "fileReplacements": [

                {

                  "replace": "src/environments/environment.ts",

                  "with": "src/environments/environment.prod.ts"

                }

              ],

              "optimization": true,





Run

Save

Reset

**Your app can be found at:**[https://ed-6434592358924288.educative.run](https://ed-6434592358924288.educative.run/)

Open chrome dev tool

When the application compiles, you can open the given link to view your application. Then right-click on the application page, click Network, and you should see the following output:

Chrome Dev Tools Network Panel

Within this section, we can see what the browser is downloading, as well as how long each file is taking to download (this helps us see if any individual files are slowing down the application). There is so much that can be inspected within this tool, so it is worth reading through the official documentation of the Dev Tools for your browser of choice. Here is a link to the [Chrome Dev Tools](https://developers.google.com/web/tools/chrome-devtools/network/reference).

# Summary: Packaging Our Application

Here is a quick summary for you!

**We'll cover the following**

* [Ahead-of-Time compilation](https://www.educative.io/courses/getting-started-with-angular/N0l506YvvDD#Ahead-of-Time-compilation)
* [Lazy loading](https://www.educative.io/courses/getting-started-with-angular/N0l506YvvDD#Lazy-loading)
* [angular.json file](https://www.educative.io/courses/getting-started-with-angular/N0l506YvvDD#angular.json-file)

In this chapter, we have explored a range of options that we can use in order to create a fast, optimized application.

In today’s world of fast internet connections, users expect any website or application they use – whether it’s at work, home, or on mobile – to load almost straight away. Thankfully, Angular provides us with a number of strategies that we can use in order to make the fastest version of the application we can.

Packaging Angular application

## Ahead-of-Time compilation

We can use Ahead-of-Time compilation in order to remove the need for the browser to compile the application. This reduces the size of the Angular framework the browser downloads and reduces the time it takes for a section to run in the browser.

## Lazy loading

We also covered lazy loading, which is a strategy where, due to the use of modules, we can get the browser to download only the Bundled JavaScript files needed to run a section of the application. If we have a bundle for each feature and a feature isn’t being used, then the bundled JavaScript file is not downloaded until it’s needed. This reduces the amount of initial downloads the browser needs to make.

## angular.json file

We spent time exploring the angular.json file, which is the main configuration file for our project. Within this file, we can make a wide variety of changes, including switching out **environment files** for different builds. Having different configuration settings for the different environments our application may run into during the entire development process is crucial.

The angular.json file can also be amended so that it has a set of final file sizes. We need to be warned if the application gets too big. Being aware of these issues before the application is released to the end-user means that we can find ways to reduce the end file size before the user finds that the application takes too long to load.

There are many ways we can tune and monitor the performance of the final production application since Angular should never be seen as a slow framework. It just takes time and testing in order to make the ideal settings for your environment.

# Installing the Angular CLI

Let's install the Angular CLI.

**We'll cover the following**

* [Step 1: Install Node](https://www.educative.io/courses/getting-started-with-angular/m2XqO6q5KO3#Step-1:-Install-Node)
* [Step 2: Check Node version](https://www.educative.io/courses/getting-started-with-angular/m2XqO6q5KO3#Step-2:-Check-Node-version)
* [Step 3: Install Angular CLI](https://www.educative.io/courses/getting-started-with-angular/m2XqO6q5KO3#Step-3:-Install-Angular-CLI)
* [Step 4: Check Angular version](https://www.educative.io/courses/getting-started-with-angular/m2XqO6q5KO3#Step-4:-Check-Angular-version)

## Step 1: Install Node

To install the Angular CLI, we need Node, so we need to go to the [Node website](https://nodejs.org/en/) and install it. Once we’re at the Node website, we need to click on the download link for the **Long Term Stable (LTS)** version of Node. This version of Node is recommended for most users and is ideal for our needs.

Node.js logo

Once the download has completed, click through the installation wizard and complete the installation of Node.

## Step 2: Check Node version

Then, when that has finished, we need to open up our Terminal or Command Prompt and check the version of Node we’ve just installed. To do this, run the following command in the terminal:

node -v

📝 **Note:** Click the terminal window and see what happens!

**Terminal 1**

Terminal

**Click to Connect...**

****

You should see the version number of Node that is installed on our machine, which confirms that the Node has been installed.

## Step 3: Install Angular CLI

OK, that’s done. Now, let’s install the Angular CLI.

Angular CLI

###### Question

Why do we need to install the **CLI** when we’ve just installed **Node**?

Show Answer

Again, in your Terminal or Command Prompt, run the following command:

npm install -g @angular/cli

This command is telling npm to go and install the Angular CLI package globally (so that it’s available within any folder). Then, npm goes off and downloads the latest version of the Angular CLI and installs it for you.

## Step 4: Check Angular version

Then, when that has finished, we need to check the version of Angular we’ve just installed. To do this, run the following command in the terminal:

ng --version

📝 **Note:** Click the terminal window and see what happens!

**Terminal 1**

Terminal

**Click to Connect...**

****

You should see the Angular and Node version that is installed on our machine, which confirms that Angular has been installed.

# Create Your First Angular App

Let's create our first Angular app using the Angular CLI.

**We'll cover the following**

* [ng new <name>](https://www.educative.io/courses/getting-started-with-angular/xoJJNPg3MKE#ng-new-%3Cname%3E)
* [Expected output](https://www.educative.io/courses/getting-started-with-angular/xoJJNPg3MKE#Expected-output)

Now, we should have the Angular CLI installed, and it’s time to create our first Angular app. This is just going to be a small app with which we can look through the code and see how an Angular app is made and what the structure of the app is.

## ng new <name>

To create an Angular application, we need to go back into our Terminal or Command Prompt and navigate to a folder where we can work. Once we have navigated to our development folder, we simply run the command ng new <name>, along with the name of the Angular app we’re building. So, for this, type in the following:

ng new angular-architecture

Click the terminal window and see what happens!

**Terminal 1**

Terminal

**Click to Connect...**

****

## Expected output

This will create a new folder within the **development** folder, called angular-architecture. CLI uses the name you provide to create a new folder in which you’ll create the application. Once that has run, CLI will prompt the user with different questions related to the application setup, and then you should see the following message:

✔ Packages installed successfully.  
    Successfully initialized git.

Congratulations! You’ve created your first Angular app.

📝 **Note:**

* Use ls to see all the files and folders within the current directory.
* Use cd <foldername> to navigate inside a subdirectory.
* Use cd.. to navigate to the previous directory.