ANGULAR ROUTER JUMPSTART

BUILD THE NAVIGATION SYSTEM OF A SINGLE PAGE APPLICATION

ANGULAR UNIVERSITY

Table Of Contents

Note: see details at the end of this book for getting the Free Angular For Beginners Course, plus a bonus content Typescript Video List.

- Configuring the Angular Router
- Configuring a primary router outlet
- Bootstrapping the router
- What could go wrong so soon?
- Understanding the problem
- Why things used to work when using hash navigation
- How to prevent 404 not found in the new router?
- Home and Fallback routes - why order matters
- Router Navigation with the routerLink directive
- Programmatic router navigation
- Route Parameters - Avoid Memory Leaks
- The notion of Route Snapshot and Router Snapshot
- Why do we a need a snapshot of the whole router?
- Implement Master-Detail using Child Routes
- Using Child Routes for implementing Master Detail
- Auxiliary Routes: what are they and when are they useful?
- Multiple outlets, but only one URL?
- Practical use cases of auxiliary routes
- An example of an auxiliary route configuration
- What does the URL look like for accessing an auxiliary route?

Section 3 - Build a Navigation Menu with Bootstrap 4

- The Menu System That We Are About To Build
- Implementing the Top Menu
- Configuring the Router Top Menu Navigation

Angular Router Jumpstart

Introduction

Welcome to the Angular Router Jumpstart Book, thank you for joining! Like the title says, the goal of the book is to get you proficient quickly in the Angular Router.

We are going to start by doing an introduction to single page applications, and then we will cover the Angular Router and some of its most commonly used features and use cases.

Then we will take that knowledge and we will apply it to a concrete use case: we will build the navigation system of a small application.

So without further ado, let's get started!

I hope you will enjoy this book, please send me your feedback at admin@angular-university.io.

Section 1 - Single Page Applications (or SPAs)

We hear the term Single Page Application a lot today. It looks like a lot of projects starting development today being built using this solution!

But if you have never used a single page application, you will probably have these questions in mind:

- Why a Single Page Application?
- What are the Benefits of the SPA approach?
- What is a SPA?

Let’s start with the first question: Why Single Page Applications (SPAs)? This type of applications have been around for years, but they have not yet become widespread in the public Internet.

And why is that?

There is both a good reason for that and an even better reason on why that could change in the near future.

SPAs did get adopted a lot over the last few years for building the private dashboard part of SaaS (Software as a Service) platforms or Internet services in general, as well as for building enterprise data-driven and form-intensive applications.

But does Angular then enforce building our applications as a SPAs?

The answer is no, it does not, but that is an interesting possibility that it brings, given the many benefits of building an application that way. Which brings us to a key question:

Why Would You Want to Build an Application as a SPA?

This is something that is often taken for granted. Everybody is building applications this way, but what is the big advantage of that? There have to be at least a couple of killer features, right?

Let’s start with a feature that does not get mentioned very often: Production Deployment.

Understanding the advantages of SPA use in production will actually also answer the key question:

What is a Single Page Application?

Sometimes names in Software Development are not well chosen, and that can lead to a lot of confusion. That is certainly not the case with the term SPA: a single page application literally has only one page !!

If you want to see a Single Page Application in action, I invite you to head over to the AngularUniv https://angular-university.io and start clicking on the home page on the list of latest courses, and on the top menu.

If you start navigating around, you will see that the page does not fully reload – only new data gets sent over the wire as the user navigates through the application – that is an example of a single page application.

Let me give you an explanation for what is the advantage of building an application like that and how does that even work.

Using the Chrome Dev Tools, let’s inspect the index.html of the AngularUniv website that gets downloaded on the home page (or any other page, if you navigate and hit refresh).

```
<meta charset="UTF-8">
<head>
    <link href="https://fonts.googleapis.com/icon?family=Material+Icons" re
    <link href="//maxcdn.bootstrapcdn.com/font-awesome/4.6.2/css/font-aweso
          rel="stylesheet" type="text/css"/>
    <link rel="shortcut icon" href="favicon.ico" type="image/x-icon"/>
    <link rel="stylesheet"
          href="https://angular-university.s3.amazonaws.com
                /bundles/bundle.20170418144151.min.css">
</head>

<body class="font-smoothing">

<app>... </app>

<script
        src="https://angular-university.s3.amazonaws.com
             /bundles/bundle.20170418144151.min.js"></script>

</body>

```

The page that gets downloaded is the very first request that you will see in the Chrome Network tab panel – that is the literally the Single Page of our Application.

Let's notice one thing – this page is almost empty! Except for the tag, there is not much going on here.

> *Note: on the actual website the page downloaded also has HTML that was pre-rendered on the server using Angular Universal*

Notice the names of the CSS and Javascript bundles: All the CSS and even all the Javascript of the application (which includes Angular) is not even coming from the same server as the `index.html` – this is coming from a completely different static server.

> *In this case, the two bundles are actually coming from an Amazon S3 bucket!*

Also, notice that the name of the bundles is versioned: it contains the timestamp at which the deployment build was executed that deployed this particular version of the application.

The Production Deployment Advantages of Single Page Applications

The scenario we have seen above is actually a pretty cool advantage of single page applications:

> *Single Page Applications are super easy to deploy in Production, and even to version over time!*

A single page application is super-simple to deploy if compared to more traditional server-side rendered applications: it's really just one `index.html` file, with a CSS bundle and a Javascript bundle.

These 3 static files can be uploaded to any static content server like Apache, Nginx, Amazon S3 or Firebase Hosting.

Of course the application will need to make calls to the backend to get its data, but that is a separate server that can be built if needed with a completely different technology: like Node, Java or PHP.

Actually, if we would build our REST API or another type of backend in Node, we could even build it in Typescript as well, and so be able to use the same language on both the server and the client, and even share some code between them.

Single Page Application Versioning in Production

Another advantage of deploying our frontend as a single page application is a versioning and rollback. All we have to do is to version our build output (that produces the CSS and JS bundles highlighted in yellow above).

We can configure the server that is serving our SPA with a parameter that specifies which version of the frontend application to build: it's as simple as that!

This type of deployment scales very well: static content servers like Nginx can scale for a very large number of users.

Of course, this type of deployment and versioning is only valid for the frontend part of our application, this does not apply to the backend server. But still, it's an important advantage to have.

But is production deployment the only advantage of single page applications? Certainly not, here is another important advantage:

Single Page Applications And User Experience

If you have ever used a web application that is constantly reloading everything from the server on almost every user interaction, you will know that that type of application gives a poor user experience due to:

- the constant full page reloads

- also due to the network back and forth trips to the server to fetch all that HTML.

In a single page application, we have solved this problem, by using a fundamentally different architectural approach:

On a SPA, after the initial page load, no more HTML gets sent over the network. Instead, only data gets requested from the server (or sent to the server).

So while a SPA is running, only data gets sent over the wire, which takes a lot less time and bandwidth than constantly sending HTML. Let's have a look at the type of payload that goes over the wire typically in a SPA.

For example, in the AngularUniv SPA, if you click on a course, no HTML will be sent over the wire. Instead, we get an Ajax request that receives a JSON payload with all the course data:

```
{
  "summary": {
    "id": 9,
    "url": "angular2-http",
    "description": "Angular RxJs Jumpstart",
    "longDescription": "long description goes here",
    "totalLessons": 15,
    "comingSoon": false,
    "isNew": false,
    "isOngoing": false,
    "visibleFrom": "1970-01-31T23:00:00.000Z",
    "iconUrl": "https://angular-university.s3.amazonaws.comangular-http-v2.
    "courseListIcon": "https://angular-academy.s3.amazonaws.com/observables
  },
  "lessons": [...]
}

```

```
18
```

As we can see, the HTML version of a course would be much larger in size when compared to a plain JSON object due to all the opening and closing tags, but also there is a lot of duplicate HTML if we constantly loading similar pages over and over.

For example, things like the headers and the footers of a page are constantly being sent to the browser but they have not really changed.

So with this, we have here a second big advantage of a single page application: a much-improved user experience due to less full page reloads and a better overall performance because less bandwidth is needed.

So Why Don’t We Use SPAs Everywhere Then?

If SPAs have so many advantages, why haven’t they been adopted on a larger scale on the public internet? There is a good reason for that.

Until relatively recently, search engines like Google had a hard time indexing correctly a single page application. But what about today?

In the past, there were some recommendations to use a special Ajax Crawling Scheme that has been meanwhile deprecated. So is Google now able to fully render Ajax?

In an official announcement, we have the information that Google search is now generally able to crawl Ajax, but there are some reports that it's yet not completely able to do so.

And the recommendation is to use Progressive enhancement instead. So what does that mean, at this stage is it possible to use SPAs in the public internet or not yet?

Search Engine Friendly SPAs?

It's possible to have the performance and user experience of a SPA together with good SEO properties: the use of the Angular Universal pre-rendering engine allows us to:

- pre-render an application on the backend
- ship the HTML to the browser together with Angular
- and have Angular bootstrap on the client side and take over the page a SPA

Will SPAs Become More Frequent In the Future?

Imagine an SEO friendly version of Amazon that would not refresh itself at each page reload and with a much-improved performance and user experience: that would have likely a huge positive impact on the time customers spend on the site!

So the technical benefits of SEO-friendly SPAs are significant and even more so on mobile, and we would expect that these type of SEO friendly SPA applications would become more frequent in the future also on the public internet.

But this leaves the last question of this section still unanswered. You must be thinking at this stage after reading the last few sections:

If only minimal HTML is loaded from the server at startup time, then how does the page keep changing over time?

Which leads us to the last question of this section, and maybe the most important:

How Do Single Page Application Even Work?

Indeed, how can they work because we only loaded very little HTML from the server? Once the application is started, only data goes over the wire. So how does the new HTML come from?

Because there has got to be new HTML being generated somewhere, as it's the only way that the browser will change what it's displaying, right?

The answer is simple, and it has to do with the way that single page applications actually work when compared to traditional server-based applications.

On a traditional application (which includes the vast majority of today's public Internet), the data to HTML transformation (or rendering) is being done on the server side.

On the other hand, Single page applications do it in a much different way:

> *In a SPA after application startup, the data to HTML transformation process has been moved from the server to the client – SPAs have the equivalent of a template engine running in your browser!*

And so with this information in hand, let’s wrap up this section by summarizing the key points about single page applications.

Section Summary – Advantages of SPAs and the Key Concept About How They Work

Building our application as a SPA will give us a significant number of benefits:

- We will be able to bring a much-improved experience to the user

- The application will feel faster because less bandwidth is being used, and no full page refreshes are occurring as the user navigates through the application

- The application will be much easier to deploy in production, at least certainly the client part: all we need is a static server to serve a minimum of 3 files: our single page index.html, a CSS bundle, and a Javascript bundle.

- We can also split the bundles into multiple parts if needed using code splitting.

- The frontend part of the application is very simple to version in production, allowing for simplified deployment and rollbacks to previous version of the frontend if needed

And this just one possible deployment scenario of SPAs in production.

Other Production Scenarios

Other scenarios include pre-rendering large parts of the application and upload everything to a static hosting server, in-memory caching on the server of only certain pages, do versioning using DNS, etc.

Its today simpler than ever to make SEO-friendly SPAs, so they will likely have increased adoption in the future, in scenarios where SPAs have not been frequently used.

The Way SPAs Work

The way that single page applications bring these benefits is linked to the way that they work internally:

- After the startup, only data gets sent over the wire as a JSON payload or some other format. But no HTML or CSS gets sent anymore over the wire after the application is running.

The key point to understand how single page applications work is the following:

> *instead of converting data to HTML on the server and then send it over the wire, in a SPA we have now moved that conversion process from the server to the client.*

The conversion happens last second on the client side, which allows us to give a much-improved user experience to the user.

I hope that this convinces you of the advantages of SPAs. If not please let me know and why that is not the case so I can better develop some of the points.

Also, at this stage, I want to show you some code straight away. Let's build a small Hello World SPA in Angular, and take it from there. We are going to see how the concepts that we have introduced map to the small application that we are about to build.

This is just the beginning so let's get going.

Section 2 - Angular Router Crash Course

In this section we are going to cover the following topics:

- Initial Router Setup and configuration, avoiding a usual setup pitfall
- Setup a home route and a fallback route, learn why order matters
- Router navigation using routerLink
- Master-Detail with Child Routes - how do Child Routes work?
- The notion of Route Snapshot and Router Snapshot
- Auxiliary Routes: what are they and when are they useful?

Configuring the Angular Router

The first thing that we should do is simply write some routing configuration. What we are doing in that configuration is to map certain URL paths to Angular components: meaning that if there is path match then the component gets displayed.

The configuration is of type `RouteConfig`, which is just an array of `Route` configuration object. Let's configure a couple of simple routes:

```
export const routeConfig:Routes = [
    {
        path: 'home',
        component: Home
    },
    {
        path: 'lessons',
        component: AllLessons
    }
];

```

This configuration simply means: if you navigate to `/home` then the `Home` component gets displayed; if you navigate to `/lessons` then `AllLessons` component gets displayed. And if you navigate elsewhere you are going to get an error.

But where do these components get displayed?

Configuring a primary router outlet

Once the router has a URL match, it will try to display the corresponding matching components. for that it ill look in the template for a `router-outlet` component:

```
@Component({
    selector:'app',
    template: `

  <main>
      <router-outlet></router-outlet>
  </main>
  `
})
export class App {
    ...
}

```

Router outlet is a dynamic component that the router uses to display in this case either the `Home` or `AllLessons` components. There is nothing special about these components, these could be any component.

Bootstrapping the router

To finish the setup of the router, we also need to add its directives and injectables into the Angular bootstrap system. We do so by adding by importing the `RouterModule` into the application root module:

```
import {RouterModule} from "@angular/router";
import {BrowserModule} from "@angular/platform-browser";
import {RouterModule} from "@angular/router";

@NgModule({
    declarations: [App],
    imports: [BrowserModule, RouterModule.forRoot(routeConfig)],
    bootstrap: [App]
})
export class AppModule {
}

platformBrowserDynamic().bootstrapModule(AppModule);

```

Notice that we configure the module by using the `forRoot` function instead of simply adding the `RouterModule`.

With this, if we access the `/home` or `/lessons` URLs, we would get the corresponding component displayed. But here is where for new users of the router sometimes things start to go wrong.

What could go wrong so soon?

With the current setup, if we navigate to an URL from the index page using the router built-in mechanisms, it all works. but if we try to type the URL in the browser address bar to access directly for example `/lessons`, we get a 404 page not found error. Why is that?

This is something that it's better to solve from the very beginning, otherwise we will not have a good router development experience, and the application won't be usable in production. Let's go over the problem.

Understanding the problem

The first thing to know about the new router is that by default it uses the HTML5 History API. This means that routing is not based anymore on using the `#` part of the URL, which is used to link directly to a section of the page. The `#` is typically used for example to link directly to a section of a blog post.

This means that when the router navigates to `/lessons`, that URL is really shown in the browser address bar. This is opposed to the ancient routing were the browser address bar would display `/#/lessons` instead.

Why things used to work when using hash navigation

In the ancient strategy the URL was pointing to the root of the domain, so when hitting refresh this would reload `index.html`, our single page application.

This is because the `#` part of the URL was ignored by the server.

But now in the new strategy, this will cause an attempt to load a file called `lessons`, which does not exist, and so we get 404 Not found.

How to prevent 404 not found in the new router?

In order to use the new HTLM5 strategy, you need to setup your server so that any unmatched request gets directed to `index.html`, so that for example `/lessons` gets as result `index.html` and not 404 Not found.

The exact configuration will depend on which server technology is being used. Let's give an example, let's say we are using Node for the server and Express as the backend web framework.

To fix the 404 issue we would have to configure the following middleware as the last in the middleware chain:

```
function redirectRouterLessonUnmatched(req,res) {
    res.sendFile("index.html", { root: './index.html' });
}

app.use(redirectRouterLessonUnmatched);


```

This would give us a good start in using the new HTML5 mode of the router, without this the router usability is compromised. Another common source of issues is, how to configure a default or a fallback route, this is probably something that all applications need.

Home and Fallback routes - why order matters

The new component router supports the notions of empty paths and wildcards, meaning that we can configure an index route and a fallback route like this:

```
export const routeConfig:Routes = [
     ... ,
    {
        path: "",
        component: Home
    },
    {
        path: "**",
        component: PageNotFoundComponent
    }
];

```

We can see that the empty path configuration would map the URL `/` to the component `Home`, and all other paths to `PageNotFoundComponent`. But there is a catch in this configuration as well.

Why order matters

One of the key things to know about routing configuration is that the order matters a lot. When the router receives an URL, it will start going through the configuration in order: from the first element of the configuration array.

If it finds a match to the complete URL, it stops and instantiates the corresponding component(s). So in this case, if we would put the fallback configuration in the beginning of the array, every URL will match to the `**` wildcard and this break routing.

That's why we should put the fallback route configuration as the last entry in the array. With this baseline configuration, let's now set up some router navigation. There are two ways of doing this:

- declarative template based navigation with the `routerLink` directive
- programmatic or imperative navigation with the `Router` API

Router Navigation with the routerLink directive

As we have included `RouterModule` in our app, we can use the `routerLink` directive to define router navigation links in our template. There are a couple of ways of doing this:

```html
<ul class="top-menu">
    <li>
        <a routerLink="">Home</a>
    </li>
    <li>
        <a routerLink="courses">Courses</a>
    </li>
    <li>
        <a [routerLink]="['lessons']">Lessons</a>
    </li>
</ul>

```

We can either hardcode a string directly in the template, like its the case of the home route or the courses route. But we can also pass it an expression. If so we need to pass it an array containing the multiple URL path parts that we want to navigate to: in this case we want to navigate to the `/lessons` path.

Programmatic router navigation

Another way of doing router navigation is to use the router programmatic API to do so. For that we just have to inject the router into our component, and make use of either the `navigate` or `navigateByUrl` navigation methods:

```

constructor(private router:Router) {

}

openCourse(course) {
    this.router.navigateByUrl(`/courses/${course.id}`);
}

showCoursePlayList(course) {
    this.router.navigate(['/courses',course.id]);
}

```

One of the things that we usually want to do when navigating between two routes is to pass navigation parameters to the target route.

Route Parameters - Avoid Memory Leaks

If we want to read parameters between routes, we probably want to use the route parameters observable that the Router API exposes. For example, when navigating to the course detail component using `/courses/1` (1 being the course Id), we can recover the course Id from the URL:

```
constructor(router: Router) {

   route.params.subscribe(
        params =>{
            this.courseId = parseInt(params['id']);
        }
    );
}

```

As we can see the router provides observables that allow us to observe routing changes and react to them. One important pitfall to avoid with router navigation is to prevent memory leaks.

Have a look at the lesson Exiting an Angular Route - How To Prevent Memory Leaks for some more details.

The notion of Route Snapshot and Router Snapshot

One key notion about the new component router is that it's reactive. This means its API exposes several observables that can be subscribed to react to routing changes.

But sometimes we don't want the latest value of a route or its parameters, we just the values that were present only at the moment when the component was initially instantiated, and we usually want those values synchronously and not asynchronously.

For that, the reactive router introduces also the notion of *snapshot*. A snapshot can be injected in the constructor of the routed component:

```
constructor(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {

    const parentRouteId = state.parent(route).params['id'];

    ...
}
```

With these snapshots, we have access to the route parameters at the moment of navigation.

Why do we a need a snapshot of the whole router?

We can access the snapshot of the current route, but also of the whole router. The snapshot of the whole router would be useful for accessing for example route parameters of parent routes, like in the example above.

And this is just one of the many new features of the new router. The new router also implements the usual features of routing, such as Child routes which allow us to implement common UI patterns like Master Detail.

Implement Master-Detail using Child Routes

Actually we were already using at least the notion of child routes without even knowing. When a route has multiple child routes, only one of those child routes can be active at any given time.

This sounds a lot like what was happening with our top-level routing configuration. Only `/home` or `/lessons` can be active at any given time. In fact, using the empty path routing feature and making the top-level route a componentless route, we can rewrite the configuration using child routes:

```
export const routeConfig:Routes = [
    {
        path: '',
        children: [
            {
                path: 'home',
                component: Home
            },
            {
                path: 'lessons',
                component: AllLessons
            }
        ]
    }

];

```

This gives the exact same result as before: only `/home` or `/lessons` can be active at one given time. Remember all this route config only has one `router-outlet`, so the end result of the matching must be a single component and not multiple.

Also, note that a componentless route is a route that participates in the path matching process but does not trigger the instantiation of a route component.

Using Child Routes for implementing Master Detail

One use case of child routes is to use them to implement the common UI pattern known as Master-Detail. Actually we are going to go one step further and implement a master route with multiple types of detail routes.

Imagine a course with a list of lessons. You click on a lesson in the list in order to display it. But there is a catch, there are multiple types of lessons: video lessons, text lectures, quizzes or interactive exercises.

One way to configure this would be to use child routes:

```
export const routeConfig:Routes = [
    {
        path: 'courses',
        children: [
            {
                path: ':id',
                children: [
                    {
                        path: '',
                        component: CourseLessons,
                    },
                    {
                        path: 'videos/:id',
                        component: VideoLesson
                    },
                    {
                        path: 'textlectures/:id',
                        component: TextLesson
                    },
                    {
                        path: 'quizzes/:id',
                        component: QuizLesson
                    },
                    {
                        path: 'interactivelessons/:id',
                        component: InteractiveLesson
                    }
                ]
            }
        ]
    }
];

```

This is one way to do it, we have gone here into several levels of nesting to show that it's possible.

The way this works is that when the user clicks in one of the lessons, depending on the link clicked a new detail screen will show replacing the master `CourseLessons` component.

This is the basis of child routes which is a common feature in many routers. Another common feature that is sometimes not used to its full potential are auxiliary routes.

Auxiliary Routes: what are they and when are they useful?

First, what are auxiliary routes? These are just plain routes like the primary route that was mapped to the `router-outlet` component. But instead, auxiliary routes are mapped to a different outlet which must be named (unlike the primary outlet).

This is an example of a page with multiple outlets, each corresponding to a subset of routing configuration:

```
<div class="main-container">
    <router-outlet></router-outlet>
    <router-outlet name="section1"></router-outlet>
    <router-outlet name="section2"></router-outlet>
    <router-outlet name="section3"></router-outlet>
</div>

```

But how can this work, because all matching is done using the URL, and there is only one URL. Right ?

Multiple outlets, but only one URL?

The key thing to realize about top-level auxiliary routes is that effectively each one has its own URL to match to, starting at `/`. Auxiliary routes can also be configured below the top-level, but let's focus here on the top-level auxiliary route scenario, as its very common.

Imagine that you divide your browser window into multiple mini-browser windows, each with its own separate URL. Then you provide separate routing configuration for each of those windows because you would want those windows to be navigated separately. Here are some examples.

Practical use cases of auxiliary routes

As you can see, different outlets correspond to different auxiliary routes. But when would you want to use auxiliary routes and why?

It's very common for applications to divide the page into multiple regions:

- the top-level menu

- the side menu that often is a subsection of the top menu
- an aside on the right maybe displaying a playlist of lessons
- popup dialogs for editing a list detail, that you want to keep upon during navigation
- a chat window that stays opened during navigation

An example of an auxiliary route configuration

Imagine that to the right of our screen, we want to add a playlist of lessons that gets different content when we navigate: It might contain the list of latest lessons, or the lessons of a given course:

```
export const routeConfig:Routes = [
    {
        path: 'courses',
        ....
    },
    {
        path: 'playlist',
        outlet: 'aside',
        component: Playlist
    }
];

```

What we have configured here is that when the path of the `aside` outlet is set to `playlist`, we are going to display the component `Playlist`. This routing can be defined independently of the primary route.

Let's see how does this work, how can the URL be used to display two URLs instead of one?

What does the URL look like for accessing an auxiliary route?

The Angular Router introduces a special syntax to allow to define auxiliary route URLs in the same URL as the primary route URL. Let's say that we want to trigger navigation and show `AllLessons` in the primary outlet and `Playlist` in the `rightAside` outlet. The URL would look like this:

`/lessons(aside:playlist)`

We can see that `/lessons` would still route the primary route to the `AllLessons` component. But inside parentheses, we have an auxiliary route. First, we have the name of the outlet to which it refers to: `aside`.

Then we have a colon separator and then we have the url that we want to apply to that outlet, in this case `/playlist`. This would cause the `Playlist` component to show in place of the `aside` outlet.

Note that you could have multiple auxiliary routes inside parenthesis, separated by `//`. for example this would define the url for a left-menu outlet:

```
`/lessons(aside:playlist//leftmenu:/some/path)`
```

And with this in place, we have at this point covered many scenarios that are commonly used while building the navigation system of an application.

Let's now put all of this in practice in the next section, as well as introduce a couple of extra scenarios.

Section 3 - Build a Navigation Menu with Bootstrap 4

In this section, we are going to learn how to use several features of the Angular Router in order to build a navigation system with multiple levels, similar to what you would find in an online learning platform or an online store like Amazon (but simpler).

We will do this step by step, the goal here is to learn how to configure the Angular Router by example and learn how to implement some of the very common routing scenarios that you will likely find during your everyday development.

Routing Scenarios Covered

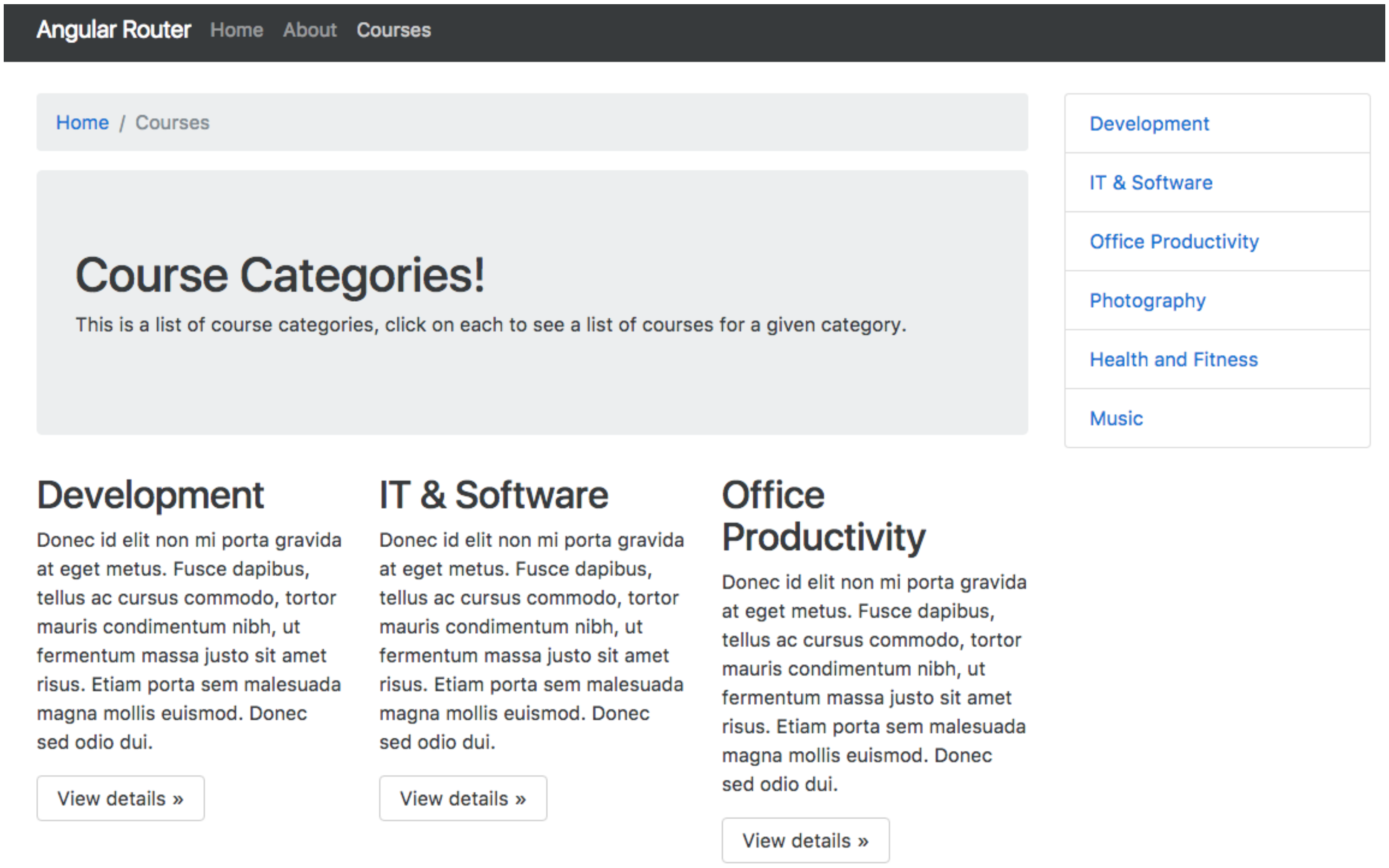
We are going to combine many features of the Router in order to build our menu, namely:

- Initial Router Setup
- Child Routes
- Nested Child Routes
- Auxiliary Routes
- Nested Auxiliary Routes

> *So let's get started building our navigation menu!*

The Menu System That We Are About To Build

This is how the menu system will look like:



Course Categories!

This is a list of course categories, click on each to see a list of courses for a given category.

Development

Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus ac cursus commodo, tortor mauris condimentum nibh, ut fermentum massa justo sit amet risus. Etiam porta sem malesuada magna mollis euismod. Donec sed odio dui.

IT & Software

Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus ac cursus commodo, tortor mauris condimentum nibh, ut fermentum massa justo sit amet risus. Etiam porta sem malesuada magna mollis euismod. Donec sed odio dui.

Office Productivity

Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus ac cursus commodo, tortor mauris condimentum nibh, ut fermentum massa justo sit amet risus. Etiam porta sem malesuada magna mollis euismod. Donec sed odio dui.

We will have the following navigation elements:

- a top menu
- A navigation breadcrumb
- A side navigation menu, that only is displayed when we navigate into `Courses`
- navigation from the courses categories list to the course category, adapting the content of the sidebar

Implementing the Top Menu

We want to start by implementing the top menu, which will always be visible no matter where we navigate inside the application. For that we are going to add the top menu at the level of the root component of the application:

```
<nav class="navbar navbar-fixed-top navbar-dark bg-inverse">
    <div class="container">
        <a class="navbar-brand">Angular Router</a>
        <ul class="nav navbar-nav" routerLinkActive="active">
            <li class="nav-item"><a class="nav-link" routerLink="home">Home
            <li class="nav-item"><a class="nav-link" routerLink="about">Abo
            <li class="nav-item"><a class="nav-link" routerLink="courses">C
        </ul>
    </div>
</nav>

<router-outlet></router-outlet>

```

If the menu is too big then a good alternative is to put it in a separate `top-menu.component.ts`. Notice the `routerLink` directives, linking to `home`, `about` and `courses`.

Also notice the `router-outlet` tag: this means the main content of the page below the top menu will be placed there. Also notice that there is no side navigation bar at this stage. The side navigation should only be visible if we click on `Courses`. Let's now write the router configuration for the top menu.

Configuring the Router Top Menu Navigation

The following initial configuration would cover the top menu scenario:

```
export const routerConfig: Routes = [
    {
        path: 'home',
        component: HomeComponent
    },
    {
        path: 'about',
        component: AboutComponent
    },
    {
        path: 'courses',
        component: CoursesComponent
    },
    {
        path: '',
        redirectTo: '/home',
        pathMatch: 'full'
    },
    {
        path: '**',
        redirectTo: '/home',
        pathMatch: 'full'
    }
];
```

As we can see, the `home`, `about` and `courses` Url paths currently map to only one component. Notice that we also configured a couple of redirects for the empty path case and a fallback route using the `**` wildcard. This is a good start, we have defined the home page, handled invalid URLs and added a couple of common navigation items.

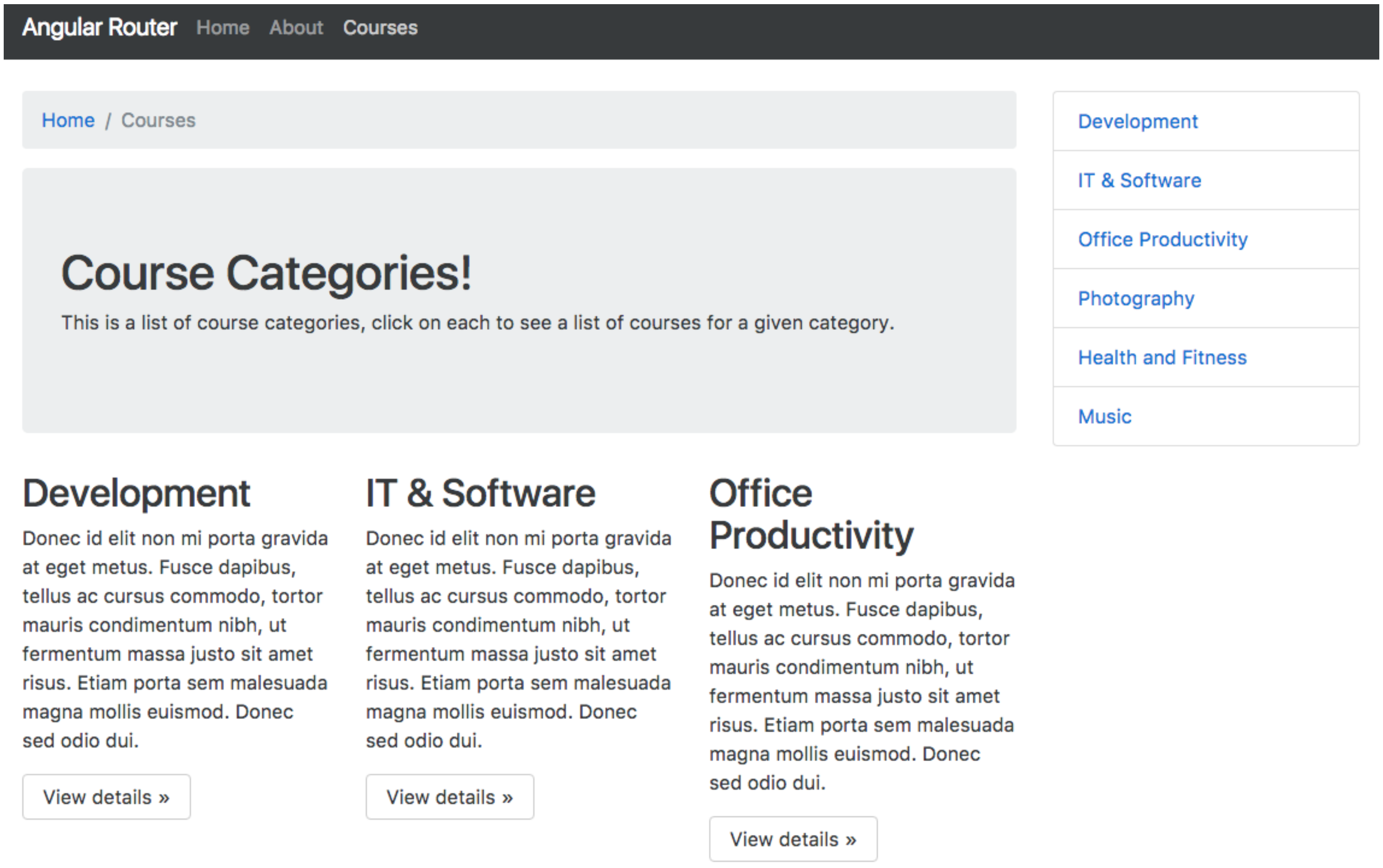
After setting up the top menu, we will now implement the following:

- we would like to show a side navigation to the user containing the list of course categories, but only if the user clicks on `Courses`

- also we would like to display a list of course category cards on the main body of the page

How will the Courses Categories page look like?

The goal is for the `CoursesComponent` to look like the following screenshot:



Course Categories!

This is a list of course categories, click on each to see a list of courses for a given category.

Development

Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus ac cursus commodo, tortor mauris condimentum nibh, ut fermentum massa justo sit amet risus. Etiam porta sem malesuada magna mollis euismod. Donec sed odio dui.

IT & Software

Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus ac cursus commodo, tortor mauris condimentum nibh, ut fermentum massa justo sit amet risus. Etiam porta sem malesuada magna mollis euismod. Donec sed odio dui.

Office Productivity

Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus ac cursus commodo, tortor mauris condimentum nibh, ut fermentum massa justo sit amet risus. Etiam porta sem malesuada magna mollis euismod. Donec sed odio dui.

There are a couple of main elements in this screen:

- there is a main 'jumbotron' with a headline "Course Categories!"
- there is a list of course category cards below, containing 3 cards per line

- there is a side menu also containing a navigation link to each course category

As we can see, the main content of the page (everything below the top menu) was applied in place of the router outlet. But this Courses page will also have other internal routing scenarios as we will see further.

Implementing a side navigation menu using a Nested Auxiliary Route

In order to create the navigation side menu, we would like the `CoursesComponent` to contain a couple of router outlets itself:

- we would need a primary outlet inside the Courses component that contains a list of course categories.
- we would need an auxiliary router outlet inside the courses component to implement the side menu.

To implement this scenario we need to first start by going over Child Routes. So how do we implement this very common routing scenario?

Implementing the Course Category Cards with a Nested Child Route

Let's have a look at the `CoursesComponent` template, to see how we have implemented it:

```
<main>
    <div class="container">
        <div class="row row-offcanvas row-offcanvas-right">
            <div class="col-xs-12 col-sm-9">
                <ol class="breadcrumb">
                    <li class="breadcrumb-item"><a routerLink="/home">Home<
```

```
                        <li class="breadcrumb-item active">Courses</li>
                    </ol>
                    <div class="jumbotron">
                        <h1>Course Categories!</h1>
                        <p>This is a list of course categories, click on each t
                    </div>
                    <router-outlet></router-outlet>
                </div>
                <router-outlet name="sidemenu"></router-outlet>
            </div>
        </div>
</main>
```

Notice that there are a couple of `router-outlet` elements inside the courses component, which is itself being injected in place of a router outlet! This is sometimes referred to as the "nested" route scenario.

We will go over the side menu outlet in a moment, but right now we want to configure the router to include the course category cards component in place of the unnamed router outlet.

Configuring the nested route scenario

In order to display the courses component and the course categories card inside it, we need the following router configuration:

```
{
        path: 'courses',
        component: CoursesComponent,
        children: [
            {
                path: '',
                component: CourseCardsComponent
            }
        ]
    }
```

This configuration means that if we hit the `/courses` URL, the following occurs:

- we should display the `CoursesComponent` in the main `router-outlet` of our application (just below the top menu)
- in place of the `<router-outlet></router-outlet>` element of `CoursesComponent` we should display the `CourseCardsComponent`

But how do we display the side menu? For that, we are going to need what is sometimes referred to an auxiliary route.

Auxiliary Routes

Auxiliary route is a common term to describe this scenario where we have multiple portions of the screen whose content is dependent upon the value of the url. In terms of router configuration, this corresponds to a route with a non-default outlet name.

The primary route is implicitly associated with an outlet without a name attribute (`<router-outlet></router-outlet>`), and after that, we can have as many other outlets in a given routing level as long as we give them different outlet names.

Implementing the side menu component

The side menu will be displayed in the `sidemenu` router outlet, and we would want the content of the side menu links to be dynamic and depend on the URL. For example, when first viewing the Courses section we would like it to contain a list of Course Categories like in the screenshot above.

But when you navigate inside a course category, we might want to display for example a list of course sub-categories.

In order to do this, the `SideMenuComponent` needs to know what the current URL is, in order to load its data depending on the URL:

```
@Component({
  selector: 'app-categories-menu',
  templateUrl: './categories-menu.component.html',
  styleUrls: ['./categories-menu.component.css']
})
export class SideMenuComponent {

  constructor(route: ActivatedRoute) {

      route.params.subscribe(params => console.log("side menu id parameter"

  }
}
```

Notice that this is just a skeleton implementation of the side menu. We would need to inject a service and get the data, or even better we could load the data before creating the component using a Router Resolver.

We can see here that the side menu component gets notified whenever a change in the URL occurs, which is the essential part of making the side menu adaptable to the URL content.

Configuring the side menu via a nested auxiliary route

We can now put the side menu in place with the following router configuration:

```
{
        path: 'courses',
        component: CoursesComponent,
        children: [
            {
                path: '',
                component: CourseCardsComponent
            },
            {
                path: '',
                outlet: 'sidemenu',
                component: SideMenuComponent
            }
        ]
    }
```

What this will do is, whenever we navigate to `/courses` we will now get the following (referring back to the file `courses.component.html` above):

- we will have `CourseCardsComponent` inside the unnamed or primary `router-outlet` element
- the new `SideMenuComponent` will be displayed inside the router outlet named `sidemenu`

We can start seeing here a pattern: a large variety of many of the routing scenarios that we typically want to implement can be configured by using only a few configuration notions: routes, outlets, child routes.

Let's see a further example of this, where we will go one navigation level deeper in the application routing!

Implementing The Course Categories Component

Let's say that now if we are in courses and click for example in `Development`, we would like the following to occur:

- we would like the content of the `router-outlet` element (the primary outlet) inside Courses to be replaced with a new component the list the contents of the course category. This could be for example a list of sub-categories

- we would like at the same time that we click on `Development` for the side menu to change its content as well, by displaying a list of links specific to the `Development` course category.

So in this scenario, we will have two different sections of the page that will react separately to the URL change: the side menu and the main body of the `Courses` page.

Configuring a new level of nesting inside the Courses page

Now let's say that the user clicks on `Development`. This will cause the URL to change to `/courses/development`. If the user clicks on `IT & Software`, the url changes to `/courses/it-software`, etc.

We want a new `CoursesCategoryComponent` component in this case to be displayed in the main body of the Courses page. For doing that, we need the following routing configuration:

```
    {
        path: 'courses',
        component: CoursesComponent,
        children: [
            {
                path: '',
                component: CourseCardsComponent
            },
            {
              path: ':id',
              component: CoursesCategoryComponent
            },
            {
                path: '',
                outlet: 'sidemenu',
                component: SideMenuComponent
            }
        ]
    }
```

Notice that we are using a `:id` path variable, to capture the URL part of the courses section. The content of this variable will be for example `development` or `it-software`.

Navigating into a Course Sub-Section

If we want to display the development section, we need in the Development card to do something like this:

```
    <div class="col-xs-6 col-lg-4">
        <h2>Development</h2>
        <p>... </p>
        <p><a class="btn btn-secondary" routerLink="development"  role="but
    </div>
    <div class="col-xs-6 col-lg-4">
        <h2>IT & Software</h2>
        <p>... </p>
        <p><a class="btn btn-secondary" routerLink="it-software" role="butt
    </div>
```

Note that this won't work in the sense that the side menu will not be notified of this navigation, more on this later.

This would display the `CoursesCategoryComponent` in the main body of the `Courses` page as expected. But what if we wanted the side menu to adapt its contents according to the navigation as well?

With the current configuration this would not happen, the side menu would still display the same content.

Making the Side Menu adjust to the current Url

In the current navigation scenario, once we get to the `Courses` page via the top menu, the side menu gets initialized and it stays in that state. In order to have the side menu also react to the URL of the current course section (like load the `development` sub-sections when clicking on `Development`), we need to modify the routing config:

```
{
        path: 'courses',
        component: CoursesComponent,
        children: [
            {
                path: '',
                component: CourseCardsComponent
            },
            // ++ THIS PART WAS ADDED
            {
              path: ':id',
              component: CoursesCategoryComponent
            },
            // -- THIS PART WAS ADDED
            {
                path: '',
                outlet: 'sidemenu',
                component: SideMenuComponent
            },
            {
                path: ':id',
                outlet: 'sidemenu',
                component: SideMenuComponent
            }
        ]
    }
```

We have now configured a navigation to occur at the level of the side menu as well. If we navigate to `/courses/development` we might expect the `SideMenuComponent` to receive the new `development` segment via the subscription to the `params` observable.

> *It turns out that won't be the case, so let's see why!*

What happened from the point of view of the side menu?

From the point of view of the side menu, we went from `/courses` to `/courses/development`, so we only affected the primary route so the content of the router outlet named `sidemenu` was kept the same.

If we want the side menu to be changed also we need to navigate in the `sidemenu` outlet as well. To understand this, try to imagine that it's like the page has multiple browser windows each with its own URL, one for the main content and the other for the side menu.

How to implement a Side Menu Navigation?

In order to trigger a navigation in the side menu, we can do it directly in the template as well. In these more advanced navigation scenarios, we can for convenience inject the router and use its navigation API to build the navigation.

You could also do this via the template, but it's great to be able to leverage Typescript auto-completion to fill in the needed navigation parameters. In these more advanced navigation scenarios auto-completion really helps as it acts like a form of documentation that is always up-to-date.

To do the navigation let's first change the HTML template of the course section card:

```
    <div class="col-xs-6 col-lg-4">
        <h2>Development</h2>
        <p>...</p>
        <p><a class="btn btn-secondary" (click)="navigate('development')"  r
    </div>
```

As we can see, we added a click handler in place of the `routerLink` directive, so we no longer configure the navigation directly in the template. Let's see how we can do the navigation programmatically.

Programmatic Navigation of Multiple Router Outlets

Let's have a look at the code, again this could have been done via the template as well:

```
@Component({
  selector: 'app-course-cards',
  templateUrl: './course-cards.component.html',
  styleUrls: ['./course-cards.component.css']
})
export class CourseCardsComponent {

  constructor(private router: Router, private route: ActivatedRoute) {

  }

    navigate(path) {
        this.router.navigate([{outlets: {primary: path, sidemenu:path}}],
                             {relativeTo: this.route});
    }

}

```

We can see that we are using the router navigation API to navigate in two outlets at the same time: the primary outlet and the `sidemenu` outlet.

This is what is happening in each outlet:

- the navigation is relative to the route that we are currently in, because we are using the `relativeTo` property and passing in this property the current active route

- in the primary outlet we are hitting the `/courses/development` URL, this should trigger the display of the `CoursesCategoryComponent` in the main `Courses` page.

- in the auxiliary `sidemenu` we are also hitting the URL `/courses/development`, which will trigger the `SideMenuComponent` to get updated according to the URL.

Remember that in the `SideMenuComponent` we have subscribed to the `route.params` observable of the active route. By now clicking on the course category of development, we will now have `development` printed to the browser console via `console.log`.

With this latest navigation logic, the side menu is now notified of the URL change. This means that the side menu could now load a different set of URLs to be displayed if needed.

What does a multiple outlet URL look like?

By triggering the programmatic navigation call above, the browser will display the following URL:

```
/courses/(development//sidemenu:development)
```

This URL means:

- the courses URL segment is active
- inside it, the primary route is set to `/courses/development`
- the auxiliary child route 'development' is active for the outlet `sidemenu`

And with this, we have the side menu functionality in place.

Section 4 - Final Thoughts & Bonus content

As we could see, The new router configuration API of the router is very powerful because it allows us to implement a multitude of scenarios by using only just a few concepts: routes, outlets and child routes.

Notice that many of the terms that we usually use to describe routing scenarios like nested routes don't actually have a direct correspondence in the routing configuration terminology.

This is is because we can obtain those scenarios by combining a couple of more generic concepts: a nested route can be obtained via a child route with a different component than the parent route, an auxiliary route is just a normal route with a non-default outlet name, other than that it has the properties of a primary route, etc.

With this small number of configuration elements we can already easily set up a lot of the navigation scenarios that we will need in just about any application: a top menu, multiple levels of navigation, side menus and much more.

At this point, you have now a rock-solid foundation on the Angular Router! You will be able to configure the router and use it in a large number of very common day to day programming scenarios.

As you could see, the Router is very powerful. With only a few concepts we can cover a vast variety of configuration scenarios.

But because the router concepts are very generic, so it's not always obvious how to map them to certain use cases, so I hope this book helped with that.

If you have any questions about the book, any issues or comments I would love to hear from you at admin@angunar-university.io

I invite you to have a look at the bonus material below, I hope that you enjoyed this book and I will talk to you soon.

Kind Regards,
Vasco
Angular University

Typescript - A Video List

In this section, we are going to present a series of videos that cover some very commonly used Typescript features.

Click Here to View The Typescript Video List

These are the videos available on this list:

- Video 1 – Top 4 Advantages of Typescript – Why Typescript?
- Video 2 – ES6 / Typescript let vs const vs var When To Use Each? Const and Immutability
- Video 3 – Learn ES6 Object Destructuring (in Typescript), Shorthand Object Creation and How They Are Related
- Video 4 – Debugging Typescript in the Browser and a Node Server – Step By Step Instructions
- Video 5 – Build Type Safe Programs Without Classes Using Typescript

- Video 6 - The Typescript Any Type - How Does It Really Work?
- Video 7 - Typescript @types - Installing Type Definitions For 3rd Party Libraries
- Video 8 - Typescript Non-Nullable Types - Avoiding null and undefined Bugs
- Video 9 - Typescript Union and Intersection Types- Interface vs Type Aliases
- Video 10 - Typescript Tuple Types and Arrays Strong Typing

Angular For Beginners Course

If you are looking to learn Angular, have a look at this free 2h introductory course.

The Angular Tutorial for Beginners is an over 2 hours course that is aimed at getting a complete beginner in Angular comfortable with the main notions of the core parts of the framework:

Click Here To View The Angular For The Beginners Course

The other Angular courses on the same website have about 25% of its content free as well, have a look and enjoy the videos.