Contents

Day01 Angular Routing and Services

PART 01 – NG16 CODE SETUP

This section assumes that you have already installed the latest Angular CLI. If you did not, please run the command **`npm install -g @angular/cli`** before proceeding. For this particular boot camp, I will use *skills* as the app folder, but you may use any other name you wish.

There is a starter file on GitHub that you may use to get started. This file is a snapshot of the code at the end of Part01, so it is NOT the beginning code, only the code after all the instructions were done in Part01.

1. From your root folder (Documents in my case), open a terminal window (or tab) to that folder and type the command **`ng new skills`**

2. Choose **N** for *stricter type* checking (if asked) and **Y** for *routing*. Choose plain **CSS** for *styles*. To choose a CSS developer, use the arrow keys on the keyboard, however the default CSS should be auto selected, just hit **Enter**.

3. Use VS Code to open the skills folder and therefore the boiler plate application. navigate to **src->app->app.component.html**.

4. Remove all the code from the app.component.html template file except the following line (around line 333):

```
<span>{{ title }} app is running!</span>
```

You may use any editor, you are not bound to VS Code

5. Also leave the custom tag for routing, so in the end you should have two lines:

```
<span>{{ title }} app is running!</span>
<router-outlet></router-outlet>
```

6. Finally for this part, add the following line to your styles.css file so that we can use Bootstrap style classes in the future:

```
@import url('https://unpkg.com/bootstrap@5.3.0/dist/css/bootstrap.min.css');
```

This should be the only line of code in this file.

PART 02 – ADDING COMPONENTS

At the point-in-time we built a boilerplate app, so in Part01, Angular installed a special module for routing. This module is called app-routing.module.ts. This is where we configure everything that concerns routing. In this part we will generate three components and configure routes to each of them.

1. In a terminal window pointing to your new folder (skills), run the following command to start a new component:

```
ng g c home --skip-tests
```

   The `--skip-tests` part is optional. It will not add a test file which is ok for this bootcamp.

2. Now we have two components, the app component that came with the boilerplate app and `home` which we just created. We can start working with the module by importing the home component into the *routing* module. This gives visibility to the `home` component and allow us to add it to the routing process:

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { HomeComponent } from "./home/home.component";
const routes: Routes = [];
```

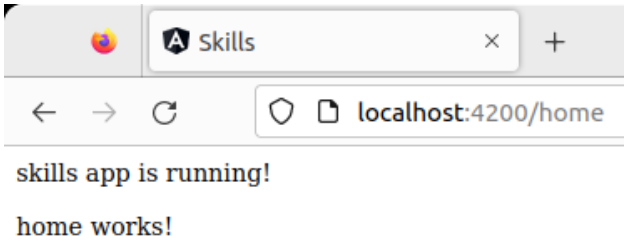
   Note: the `home` component will have its own folder as would any component we create using the `g c` command. This change is in the app-routing.module.ts file

   With that just configure the `Routes` array with individual routable objects:

```
import { HomeComponent } from "./home/home.component";
const routes: Routes = [
  {path:'home', component:HomeComponent}
];
```

   This is the very minimum required for this object.

3. If you now run the Angular command `ng serve` then go to `localhost:4200/home`, you will see the default `home` component in it's rendered form:



   This path is actually showing two components, but we will fix that issue shortly.

4. Typically, it is normal to have a default path, then add other paths as necessary:

```
const routes: Routes = [
  { path: '', redirectTo: '/home', pathMatch: 'full' },
  { path:'home', component:HomeComponent}
];
```

Here the default / path is redirected to `home`, so `home` actually is the default. The `pathMatch` property ensures no mistakes. Appendix has more on this.

5. Right now, the router is also showing content from app.component.html. Typically, this file should only contain the following code and possibly a menu:

```
<router-outlet></router-outlet>
```

We will use `router-outlet` again in Part07. Remove the `<span>` tag from app.component.html file.

6. Follow the same steps and create two more components called `register` and `login`. Then configure the `Routes` array to be able to navigate to these components. Basically repeat steps 1-3 changing the component appropriately.

PART 03 – NAVIGATION BAR AND PAGE-NOT-FOUND

1. We now have three components that we can switch between. We can formalize this relationship using a navigation bar. However, for this boot camp, we will simply add all our navigation HTML code in the app-component.html file:

```
<nav class="navbar navbar-expand-sm justify-content-center bg-light">
  <ul class="navbar-nav h5">
    <li class="nav-item"><a class="nav-link" routerLinkActive="active"
routerLink="/home">home</a></li>
    <li class="nav-item"><a class="nav-link"  routerLinkActive="active"
routerLink="/register">register</a></li>
    <li class="nav-item"><a class="nav-link"  routerLinkActive="active"
routerLink="/login">login</a></li>
  </ul>
</nav>
<router-outlet></router-outlet>
```

Notice that we use the `routerLink` directive for the final routing. If we used anchor tags, the page would reload.

2. Now in the `app-routing` module, configure all routes as objects. You may have done this in Part 02:

```
import { LoginComponent } from "./login/login.component";
const routes: Routes = [
  { path: '', redirectTo: '/home', pathMatch: 'full' },
  { path:'home', component:HomeComponent},
  { path:'register', component:RegisterComponent},
  { path:'login', component:LoginComponent},
];
@NgModule({
```

   Remember to import the `register` and `login` components at the top.

3. Although it is possible to use `ng g c` to generate a component in the usual way for a 404 error page, we can just create a .ts file and use that instead.

4. Use the editor to create a TypeScript file. If you are using VS Code, you can just right click on the app folder and choose to create a new file. Name it not-found.component.ts or something similar.

5. Just copy any of the other .ts files you have and remove the parts of the code as shown below:

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

@Component({
  selector: 'app-home',
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.css']
})
export class HomeComponent implements OnInit {

  constructor() { }

  ngOnInit(): void {
  }

}
```

6. Now change the class name and the selector to something appropriate:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-not-found',
})
export class NotFoundComponent  {

}
```

You could add CSS if you want

7. With this approach we do not have a templateUrl property, instead we write the HTML directly into the template property:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-not-found',
  template: `<h2>Oops! We can't find that resourece!</h2>`
})
export class NotFoundComponent  {

};
```

You could add CSS if you want. Note the backticks, NOT single apostrophe!

8. Now in the `app-routing` module, make sure that the `not-found` component is imported and configured last:

```
import { LoginComponent } from "./login/login.component";
import { NotFoundComponent } from './not-found.component';
const routes: Routes = [
  { path: '', redirectTo: '/home', pathMatch: 'full' },
  { path:'home', component:HomeComponent},
  { path:'register', component:RegisterComponent},
  { path:'login', component:LoginComponent},
  { path:'**', component:NotFoundComponent}
];
@NgModule({
```

Remember to import the not-found.component.ts file at the top

9. We could add just a bit of styles to the navigation at this point. In the styles.css file, add the following:

```
 @import url('https://unpkg.com/bootstrap@5.3.0/dist/css/bootstrap.min.css');
.active {
  font-weight: bold;
  background-color:bisque;
}
nav a:hover {
  background-color:cornsilk;
}
```

PART 04 – PROGRAMMATIC NAVIGATION

Other than the menu, we can navigate around the site using the built in navigation classes that Angular provides. It may seem like overkill in these examples but just for demonstration purposes, here is how programmatic navigation works.

1. Let's say we wanted to open the `login` view from the `home` view. Well, we add a method in the component we want to navigate from. In the method we add code that leads to the destination component (login). We need the `Router` class for this maneuver. Since `Router` is a service, we can inject that service via the constructor in the `home` component:

```
  export class HomeComponent implements OnInit {
  constructor(private router:Router) { }
  ngOnInit(): void {
  }
```

   The `Router` module will be imported automatically by VS Code, if not, add it manually.

2. Add the navigation method and add the destination path inside of an array:

```
  ngOnInit(): void {
  }
  goToLogin(){
    this.router.navigate(['/login']);
  }
```

   If you use the `navigate()` method, you must pass the absolute path as a parameter inside of an array. The array becomes important for passing segments to a path.

3. If you did not want to pass segments, then use `nagivateByUrl` instead:

```
  goToLogin(){
    //this.router.navigate(['/login']);
    this.router.navigateByUrl('/login');
  }
```

4. There is yet another way, using the `relativeTo` property of the `NavigationExtras` module. First import the `ActivatedRoute` class:

```
import { Component, OnInit } from '@angular/core';
import { Router, ActivatedRoute } from '@angular/router';
@Component({
```

10. Inject the `ActivatedRoute` into the component via the constructor:

```
export class HomeComponent implements OnInit {
  constructor(private router:Router, private activatedRoute:ActivatedRoute) { }
  ngOnInit(): void {
```

11. Now change the code to reflect that we are using relative paths and the `relativeTo` property:

```
  goToLogin(){
    //this.router.navigate(['/login']);
    //this.router.navigateByUrl('/login');
    this.router.navigate(['/login'], {relativeTo:this.activatedRoute});
  }
```

12. Now change the template code by just adding a button and pointing it to the method from #11, this is event binding:

```
<p>home works!</p>
<div>
  <button type="button" (click)="goToLogin()">Login</button>
</div>
```

Test by clicking the button on the `home` view, it should navigate the browser to the `login` view.

PART 05 – ROUTES WITH PARAMETERS

In this *rest*ful JavaScript world, it is common to pass parameters via the URL of the browser. For example, if you go to just `jsonplaceholder.typicode.com` you will get to the home page of that site. On the other hand, if you typed into the browser URL bar the following: `https://jsonplaceholder.typicode.com/posts` then you will see all the *posts* that the site has accumulated. If you add a 6 to the end of posts, so *posts/6* then you get the sixth post. At this site you can go further. If you entered `https://jsonplaceholder.typicode.com/posts/6/comments` then you get all the comments related to post #6.

What we will do in this boot camp, is create three administrative users and re-direct them to the admin section of the site.

1. Make the following change to the tsconfig.json file so that we can import a .json data file and work with its contents:

```
    "module": "es2020",
    "lib": [
      "es2018",
      "dom"
    ],
    "resolveJsonModule": true
  },
  "angularCompilerOptions": {
```

The *tsconfig* file is in the root folder of your project. This NOT recommended in a live application. I am using a local .json file for learning purposes here.

2. In your downloaded files for today, locate a file called db.json and copy that file into your app folder. Now you can import this mock data file into the login.component.ts file. Make sure the path is configured properly:

```
import { Component, OnInit } from '@angular/core';
import * as employeeData from "./../db.json";
@Component({
```

Now we have three employees who have admin rights to enter the administrative part of the site. We will come back to this file soon.

3. Since we are trying to capture extra bits of data in the URL, we must configure our code to work with this extra information. We want that when someone logs in with an *id*, we can get the name from our *json* file and confirm that she is an administrator. We could use the existing setup, but that could lead to problems. We need a separate path in the app-routing.module.ts file to accommodate parameters:

```
 { path:'register', component:RegisterComponent },
 { path:'login', component:LoginComponent },
 { path:'login/:id', component:LoginComponent },
];
@NgModule({
```

4. The destination component is the *login* component, so start by importing the necessary modules, `ActivatedRoute` and `Router` into login.component.ts:

```
import * as employeeData from "./../db.json";
import { ActivatedRoute, Router } from "@angular/router";
@Component({
```

These classes will help us parse the URL

5. Now add a constructor and inject both services via the constructor:

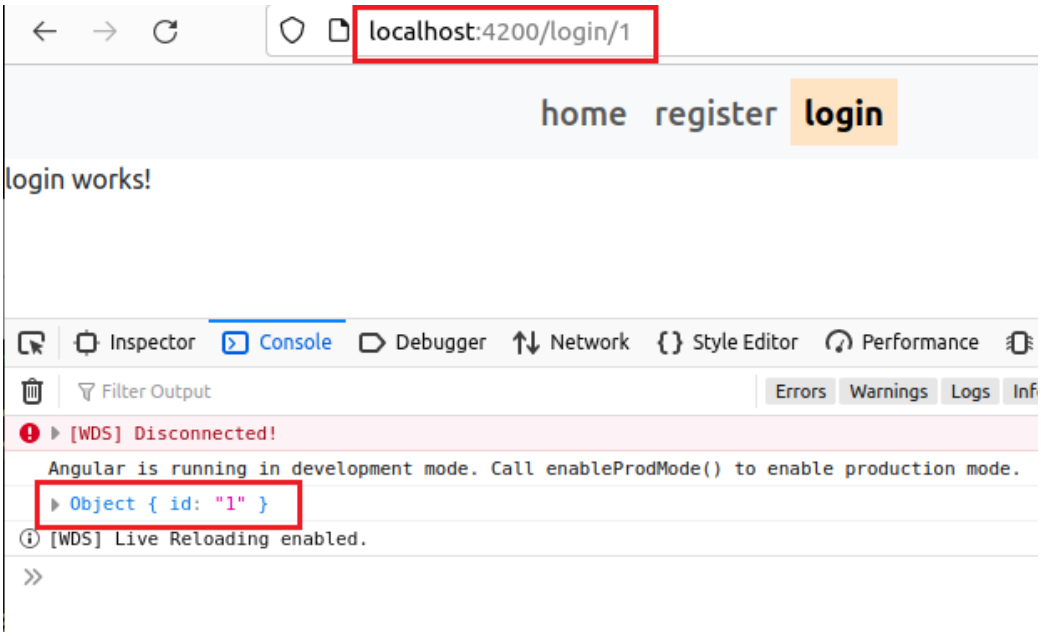
```
  styleUrls: ['./login.component.css']
})
export class LoginComponent implements OnInit {
  constructor(private route:ActivatedRoute, private router:Router ) { }
  ngOnInit(): void {
```

6. We can use the `ngOnInit()` method to accept and display any parameters being passed via the URL. The Router that we imported in #4 has a *params* property that we must *subscribe* to in order to get access to parameters:

```
ngOnInit(): void {
   this.route.params.subscribe(p => {
      console.log(p);
   })
}
```

You must import the `OnInit` module and implement the `OnInit` interface, see #5. Notice that when we subscribe to the `params` property, we supply a function to handle the actual parameter. In this case the *p* will hold the actual URL parameter.

7. Test both situations, one where we just go to `/login` and another where go to `/login/1`



As you can see, the `id` is an object!

8. We now need to work with the mock *employees* database. There are multiple ways to handle this data but here is a simple way. Declare variables to match the shape of our db.json content:

```
export class LoginComponent implements OnInit{
  constructor(private route:ActivatedRoute, private router:Router ) { };
  empID : string = "";
  name : string = "";
  ngOnInit(): void {
```

9. Remember from #6, our `p` is really an object. If we wanted the *id* part of that object we must code it like an array:

```
ngOnInit(): void {
   this.route.params.subscribe(p => {
      console.log(p["id"]);
   })
}
```

If in the URL you pass `login/2`, you should get `2` being printed out in the console window. Previously we got an entire object.

10. Declare a new property that will represent the data from the json file:

```
  empID : string = "";
  name : string = "";
  empData : any = (employeeData as any);
```

11. We can now access our data in this way:

```
ngOnInit(): void {
  this.route.params.subscribe(p => {
     //console.log(p["id"]);
     console.log(this.empData.employees);
  })
```

This would show an array.

13. This means that we can access individual elements this way:

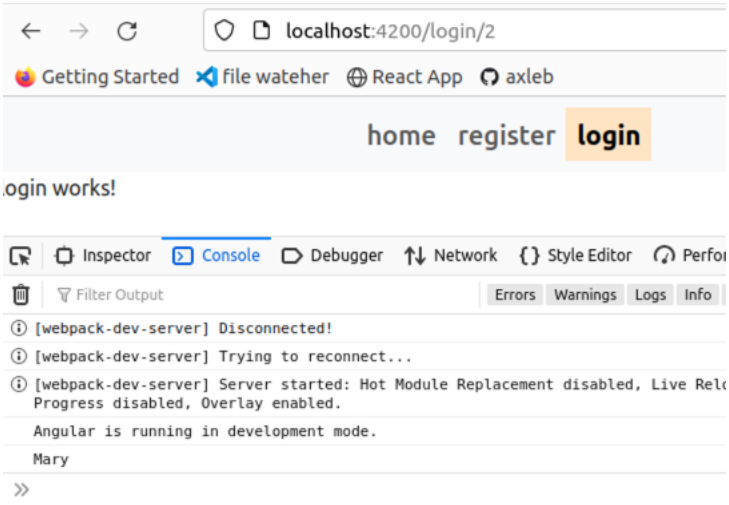
```
ngOnInit(): void {
  this.route.params.subscribe(p => {
     //console.log(p["id"]);
     console.log(this.empData.employees[0]);
  })
```

This would show Axle's data. Remember arrays are zero-based.

14. Let's now connect the *employees* data to our parameter. If we pass our `p["id"]` into our database, we can extract *username* belonging to that record:

```
  this.route.params.subscribe(p => {
    //console.log(p["id"]);
    console.log(this.emp.employees[p["id"]]?.username);
})
```

Note, the `employees[p["id"]]?.username` can be null or undefined at this point. We therefore use the **?** to mark the property as optional. The ?. is known as the *safe navigation operator*.



Note, you must pass either 0,1 or 2 as a parameter via the URL.

With this information, we can now get the `username` from the database and test it against the name that the user gave us in the login form.

15. To finish up this section, we could now pass our database value into the local name property and print it:

```
    this.route.params.subscribe(p => {
      //console.log(p["id"]);
      this.name = this.emp.employees[p["id"]]?.username;
      console.log(this.name);
   })
```

You could print this name now on the view (browser/template) using:
`Welcome {{name}} :)`

Note that the `p["id"]` will not correspond directly with our Employees since the Employees array is zero based. So in reality we should minus 1 some where. See #5 in Part 6 below.

Check with Appendix B to see how to use the other parameter methods

PART 06 – PAGE NAVIGATION

In this section we will start configuring page navigation. By Page Navigation I mean moving from one view to some other view. It does not mean turning the page if you have a lot of content that spans multiple pages. However you could do this as well.

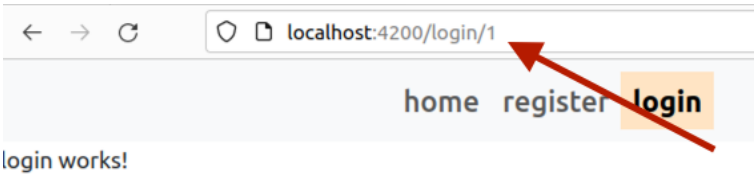
1. Change the `home` component template file to the code below. We just added a new page link at the bottom of the button and added some Bootstrap classes to create some space:

```
<div class="container">
  <div>
    <button type="button" (click)="goToLogin()">Login</button>
  </div>
  <div class="mt-4">
    <a routerLink="active" routerLink="/login">login</a>
  </div>
</div>
```

Test the functionality after adding this new link to see that it works. With `routerLink`, you do not need to inject the Router `module`.

2. In Part5 #8, we added a property, in the login.component.ts file to hold an `id` that will corelate with the id of our db.json file of employees. Let us tweak the existing `routerLink` on the `home` component, to login someone based on their id:

```
<div class="mt-4">
    <a routerLink="active" routerLink="/login/1">login</a>
</div>
```



3. A better way to accomplish the same task, is to use the following syntax:

```
<div class="mt-4">
    <a routerLink="active" [routerLink]="['/login', 1]">login</a>
</div>
```

This means that in the future we can pass the `id` of the admin into this link. We would then replace the literal number 1 with the variable.

4. To see the Administrator's name in the console window, make sure you do not have the `log()` method commented in the login.component.ts file. The `ngOnInit()` method in that file should look like the code below:

```
  ngOnInit(): void {
    this.route.params.subscribe(p => {
      this.name = this.empData.employees[p["id"]]?.username;
      console.log(this.name);
   })
  }
```

5. You will notice that the incorrect Admin is being logged in. This is because we are simply working with array indexing. We should be using array methods to search the individual objects within the `employees` array. Change the logic in login.component.ts file:

```
export class LoginComponent implements OnInit {
  empID : string = "";
  name : string = "";
  empData : any = (employeeData as any);
  constructor(private route:ActivatedRoute, private router:Router ) { }
  ngOnInit() : void{
    this.route.params.subscribe(p => {
      this.empData.employees.find();
    })
  }}
```

We can now begin to work with the `employees` object which is our database. Since that `employees` object has a `find()` method, we will use it to search for our employee. Recall we created the `empData` property in Part 05 #10.

6. The `find()` method takes a callback function that handles the result of the find:

```
      this. empData.employees.find(()=>{

      });
```

7. The parameter of the callback function needs a placeholder into which the found object goes:

```
      this. empData.employees.find((empObj:any)=>{

      });
```

For now, we type the object to `any`

8. Inside of the `find` method, we must add logic to compare our search item. For now let's search for the object with `id` of 1, this is our Administrator Axle:

```
      this.empData.employees.find((empObj : any)=>{
        if(empObj.id == "1"){
          console.log(empObj.username);
        }
      });
```

9. If you log `empObj.username`, you will see the name of Axle. However we need to find **any** of the objects. Recall that in Part 5 #9 we know that our `id` is in the reference `p["id"]`, so lets replace our literal 1 with this variable:

```
    this.route.params.subscribe(p => {
      this.empData.employees.find((empObj : any)=>{
        if(empObj.id == p["id"])
          this.name = empObj.username;
      });
      console.log(this.name);
    })
```

Here we use the property `name` and assign the found object's `username` to it.

10. (Optional) You can now change the home view to login individual administrators:

```
    <button type="button" (click)="goToLogin()">Login</button>
  </div>
  <div class="mt-4">
    <div><a routerLink="active" [routerLink]="['/login/1']">Login Axle</a></div>
    <div><a routerLink="active" [routerLink]="['/login/2']">Login Jane</a></div>
    <div><a routerLink="active" [routerLink]="['/login/3']">Login Mary</a></div>
  </div>
```

Verify that all links work. So now we don't need arithmetic to find specific data.

11. (Optional) Note, these two links will NOT work if we used the `snapshot` property:

```
  constructor(private route:ActivatedRoute, private router:Router ) { }
  ngOnInit(): void {
    this.name = this.route.snapshot.paramMap.get("id");
  }
```

If you did this, you will see the URL updated, but the name on the view will not update. This is because the `ngOnInit()` method is called just **once** for this view.

PART 07 – QUERY PARAMETERS

Query Parameters (query strings) are a way to pass key-value pairs into your code. Here is an example: http://localhost:4200/login?admin=Axle&edit=false Note this is slightly different from Route Parameters (Part 05 and 06). In Route Parameters, the parameter is **part of the route**. In Query Parameters, the parameter is **optional**.

There are three methods available to pass a Query Parameter to Route. Using `routerlink` directive, the `router.navigate` method or the `router.navigateByUrl` method.

1. Change the following code in home.component.html file to pass query parameters:

```
  <div class="mt-4">
    <div>
      <a routerLink="active"
        [routerLink]="['/login']"
        [queryParams]="{ id : 1,  edit : 0 }">
        Login Axle
      </a>
    </div>
    <div><a routerLink="active" [routerLink]="['/login/2']">Login Jane</a></div>
```

In JS a 0 translates to *false* and 1 will return *true*. So in the URL we can pass weather the admin has *edit* privileges or not. In this case only Axle has editable access. Note also, that I constructed the anchor tag vertically for space. If you hover over the login Axle link on the home view, it will show the query parameters but the other two links show only route parameters.

2. Remember in Part 04 #2 we added the `goToLogin()` method in the `home` component. With the `router.navigate()` method, we can construct query parameters in the `home` component via that method. This will be an alternative to #1 above:

```
goToLogin(){
  this.router.navigate(
    ['/login'],
    { queryParams: { id : 2, edit : 0} }
  );
}
```

The above code should produce something like this in the URL bar: http://localhost:4200/login?id=2&edit=0 You can verify this by clicking on the button on the `home` page.

3. (Optional) Just as an FYI, here is the `router.navigateByUrl()` method:

```
this.router.navigateByUrl('login?id=2&edit=0');
```

We will NOT be using this method.

4. Since we are pointing to the `login` view, lets configure that component to now accept query parameters:

```
  ngOnInit(): void {
    this.route.queryParams.subscribe(p => {
      this.empData.employees.find((empObj : any)=>{
        if(empObj.id == p["id"])
          this.name = empObj.username;
      });
    });
    console.log(this.name);
  }
```

   Change the `params` property to `queryParams`. The subscription remains the same.

5. Remove the body of the subscription method and replace it with this line:

```
ngOnInit(): void {
      this.route.queryParams.subscribe(p => {
        this.empID = p['id'] || 0;
      });
}
```

   Here we access the `id` parameter instead of an object. The `||0` is just good programming to assign a value if one is not found.

6. In the same way we need to access the second parameter via the `p` array. First, we should create a component property to store the value of the object being passed in from the URL:

```
export class LoginComponent implements OnInit {
  empID : string = "";
  name : string = "";
  pageEdit : number = 0;
  constructor(private route:ActivatedRoute, private router:Router ) { }
```

   Remember we have two values coming across, the employee's id and weather she has editable authorization or not.

7. Now we can do the assignment:

```
ngOnInit(): void {
    this.route.queryParams.subscribe(p => {
      this.empID = p['id'] || 0;
      this.pageEdit = +p['edit'] || 0;
      console.log(this.empID + "-" + this.pageEdit);
    });
}
```

   Note, the `+` in front of `p` in the fourth line is TS. This is TypeScript's way of coercing a variable to be of a certain type, here a `number`. This is the same as casting. The URL will come through as … `login?id=2&edit=0` if you click the Login button on the `home` view.

   We can cast the `id`, `edit` or both, in this way. You can now plug in the logic to identify the admin and allow or deny edit privileges. We won't be doing that part in this boot camp. Note, I have configured **only** the Login button to use this method.

PART 08 – CHILD ROUTES

A child route also uses a path and a component. However, they are part of some parent route and are inside of a separate children array. On Day02 we will see a different approach to what is shown here.

With child routes, you can have views that the user should not be able to access unless they visit through some other view. This feature is used when there is some type of major-minor relationship. An example is product-detail where there is a list of products on a component. When the user clicks on a particular product, a details view opens that gives details about the product she clicked on.

1. Start by creating a new component strictly for admin purposes on the site:

```
ng g c admin --skip-tests
```

   Do this at a terminal window. Remember to stop and restart angular.

2. Change the following code in admin.component.html file to simply add some space and make the `admin` view stand out a bit:

```
<div class="mt-4">
  <h2>admin works!</h2>
</div>
```

3. The main event for this part is in the app-routing.module.ts file. The `login` component and view will now act as a parent to the `admin` component and view:

```
const routes: Routes = [
  { path: '', redirectTo: '/home', pathMatch: 'full' },
  { path:'home', component:HomeComponent },
  { path:'register', component:RegisterComponent },
  { path:'login', component:LoginComponent, children:[
    { path:':id', component:AdminComponent },
  ] },
];
```

   The `/login` path now has a second key/value entry which is `children`. The previous `login/:id` is now part of the children array. Also, the `login/` part of the path is gone. At the same time, we point to the `AdminComponent`. Remember to import the `AdminComponent` you created in #1.

4. Since the `login` component will now work as a parent to the `AdminComponent` component, we need to add a `router-outlet` to this view. This will ensure that the path works. Do this in login.component.html file:

```
<p>login works!</p>
<router-outlet></router-outlet>
```

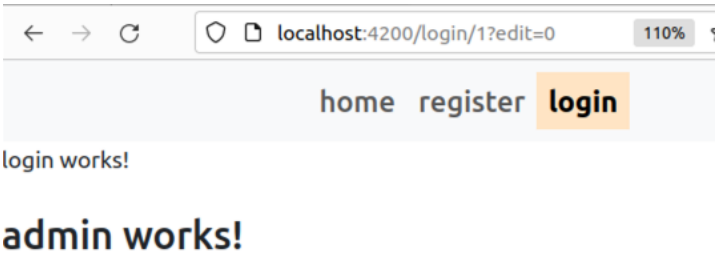
   When the app was created initially the parent component for **all** other components was app.component.html. But this will not work for *grandchildren* routes.

5. To test this, change the login for Axle on the `home` view, so home.component.html:

```
<a
  routerLink="active"
  [routerLink]="['/login', 1]"
  [queryParams]="{ id : 1, edit : 0 }">
    login Axle
</a>
```

6. Here I removed the `id` name value pair from `queryParams` and added a second entry into the `routerLink` array.

7. If you now click on the login Axle link, you see the `admin` view as a child of `login` view:



Note, the Login button and the login link on the home view, will not show the Admin component since it is not configured to show this component. Only administrators get to see the `admin works!` view:

Appendix A – Angular Architectural Concepts

Angular uses the concept of modules (Ng Modules) into which components are placed. There are built-in modules that come with the installation of Angular. Some of these modules we will be using in the course include the `HttpClientModule` and the FormsModule. An Ng Module is just a TypeScript class with an @NgModule decorator. Most decorators add metadata to the class and in come cases functionality. By default we get the AppModule to help us kickstart our customized development.

Decorators may contain declarations, exports, imports, providers and bootstrap classes. Declarations handle views like component views and directive views. Export classes ensure that a class can be accessed by other classes. Imports exposes modules required by a class. Providers handle Services which are mostly logic required by some class. Bootstrap is in the root component and provides the initial view.

There are several JS modules used as libraries in an Agular application. Libraries such as @angular/core, @angular/router and Material are used to add functionality. These libraries are simply imported.

Components comprise of a TypeScript class, some kind of HTML template for display and a stylesheet. A component will have the @Component decorator to define it as a component.

A customized component will usually have a selector which is an instructor to Angular to insert this particular component where ever it finds the selector. The selector tag within the HTML is usually written as <app-root></app-root>.

The templateUrl will point to an html file which acts as the template for a component. styleUrls of course does the same for CSS files.

Directives:

Directives are instructions that instruct the DOM as to how to place your components and business logic in the Angular project. Directives are just JS class which are declared as @directive. There are 3 directives in Angular: Component Directives, Structural Directives and Attribute Directives.

Component Directives look like this @Component. They contain the detail of how the component should be processed, instantiated and used at runtime.

Structural directives start with a * sign. These directives are used to manipulate and change the structure of the DOM elements. For example, *ngIf and *ngFor.

Attribute directives are used to change the look and behavior of the DOM elements. For example: ngClass, ngStyle etc.

The main building blocks of Angular are:

- Modules
- Components
- Templates
- Services
- Metadata
- Directives
- Data binding
- Dependency injection

Here are a few Angular CLI commands that we will be using

add	Used to add support for an external library to your project.
build	Will compile an Angular app into an output directory named dist/ at the given output path.
generate	Generates and possibly modifie files based on a schematic.
new	Creates a new workspace and a boilerplate Angular app.
run	Runs an Architect target
serve	Builds and serves your app via http, also re-compiles when it detects changes.
test	Executes unit tests in a project
update	Updates your application and its dependencies

Angular 14 File Explanation

- src folder: all the action takes place here
- app folder: all the files, that support app components.
- app.component.css: the cascading style sheets code for your app component.
- app.component.html: the template html file connected to app component and is used by angular to do any data binding.
- app.component.spec.ts: use the command ng test to see this file in action. It is a unit testing file related to app component. All files that have .spec in the middle is a test file
- app.component.ts: probably the most important typescript file which contains the view logic driving the component.
- app.module.ts: a file which includes all the dependencies for the entire website. This file defines any modules to be imported, components to be declared and the main component to start the app
- karma.config.js: This file specifies the config file for the Karma Test Runner, Karma has been developed by the AngularJS team which can run tests for both AngularJS and Angular 2+

- main.ts: As defined in angular.json file, this is the main ts file that will first run. This file bootstraps (starts) the AppModule from app.module.ts , and it can be used to define global configurations.
- polyfills.ts: This file is a set of code that can be used to provide compatibility support for older browsers. Angular 7 code is written mainly in ES6+ language specifications which is getting more adopted in front-end development, so since not all browsers support the full ES6+ specifications, pollyfills can be used to cover whatever feature missing from a given browser.
- styles.css:/ This is a global css file which is used by the angular application.
- tests.ts: This is the main test file that the Angular CLI command ng test will use to traverse all the unit tests within the application and run them.
- tsconfig.json: This is a typescript compiler configuration file.
- tsconfig.app.json: This is used to override the tsconfig.json file with app specific configurations.

tsconfig.spec.json: This overrides the tsconfig.json file with app specific unit test configurations

APPENDIX B – PARAMETERS CONTINUED

Continuing from Part05, here are other ways of accessing parameters.

1. Using snapshot.params:

```
export class LoginComponent implements OnInit {
  name : string = "";
  constructor(private route:ActivatedRoute, private router:Router ) { }
  ngOnInit(): void {
    this.name = this.route.snapshot.params["id"];
    console.log(this.name);
  }
}
```

2. Using route.snapshot.paramMap.get():

```
export class LoginComponent implements OnInit {
  name : string = "";
  constructor(private route:ActivatedRoute, private router:Router ) { }
  ngOnInit(): void {
    this.name = this.route.snapshot.paramMap.get("id");
    console.log(this.name);
  }
}
```

3. Using paramMap as an Observable:

```
export class LoginComponent implements OnInit {
  name : string = "";
  constructor(private route:ActivatedRoute, private router:Router ) { }
  ngOnInit(): void {
    this.route.paramMap.subscribe((params: ParamMap) => {
      this.name = params.get("id");
      console.log(this.name);
    });
  }
}
```

APPENDIX C – PATHS AND ROUTES IN ANGULAR

Here is a valid URL based on the Skillsoft web site.

https://	www.skillsoft.com	/codecademy/skills-training

This address is purposely split into the protocol, domain and path. In this bootcamp we will focus on the path.

1. Paths, in terms of the how the web works, may contain parameters like in the example below:

```
/login?user=Axle&auth=0
```

The receiving component can be configured to extrapolate the value of user and the value of auth and these name/value pairs may mean something.

2. Paths can also contain fragments. Here when the browser navigates to the admin view, inside that view there will be a section for user1. The browser will automatically scroll to this identifier on the view:

```
/login/admin#user1
```

Note: query parameters and fragments are not factored into Angular's calculation of routes. They can of course be used once the route or path is resolved.

3. Angular considers URLs in terms of **segments**. The segments of /login/admin/superadmin will be login, admin and superadmin.

4. The router in Angular forms a *tree* structure having just one root node. Of course there can be children and grandchildren nodes. With such a structure, Angular will attempt to resolve an entire path by working from the root path forwards. If it cannot match each part of the path, the resolving process will fail. This is where the 404 page error comes in.

5. In the case of /login/admin/superadmin the router will fail if it finds only /login/admin or /login/admin/superadmin/axle. The match has to be exact.

6. The pathMatch property attempts to resolve a path based on two values: full or prefix. The value prefix is the default.

   Using prefix, the router checks all URL elements from the left to right and stops when there is a match. If no match the router will look for a 404 error view.

   For example, '/languages/3/javascript' matches the prefix 'languages/:level' if one of the route's children matches the segment 'javascript'.

   That is, the URL '/languages/3/JavaScript' matches the config {path: 'languages/:level', children: [{path: ':javascript', component: JavaScript}]} but does not match when there are no children as in {path: 'languages/:level', component: Angular}.

   The path-match strategy 'full' matches against the entire URL. The full value is especially important in redirects. The problem here is that if full is not used, any partial matches will cause a redirect. This may not be the wanted behavior.