

Contents

PART 01 – BUILDING PROFILE SERVICE AND LOGIN FORM	2
PART 02 – IMPLEMENT PROFILE SERVICE	4
PART 03 – ROUTE GUARDS	6
PART 04 – INJECTING PROFILE SERVICE.....	7
PART 05 – LAZY LOADING MODULES	9
PART 06 – PASSING DATA AND RESOLVERS.....	12
PART 07 – REALISTIC USE OF RESOLVERS	14
PART 08 – ROUTE DATA VIA @INPUT.....	17
APPENDIX A – JSON SERVER	19

Day02 Angular Routing and Services

PART 01 – BUILDING PROFILE SERVICE AND LOGIN FORM

Today we will require the **json-server**. The instructions to install this server is in the Appendix. With the json-server set up, we can use that to create a mock API service. The JSON server is its own server, operating on an independent port. We can access that server's utilities via API calls. It is these API calls that will provide our Angular service with its content.

Create a new Angular *service* called profile: `ng g s profile`

We will use this user profile service for all of our authentication needs. For now, we only need login, logout and check if the user is logged in or not (status).

We will be using the Reactive Forms Module for this part and most of this code was developed in a previous boot camp.

1. Create all three methods in the ProfileService class:

```
export class ProfileService {  
  constructor() { }  
  login() {}  
  logout() {}  
  status() {}  
}
```

2. The service will be very simple for now, until we integrate the json-server:

```
export class ProfileService {  
  login() {  
    return true;  
  }  
  logout() {  
    return false;  
  }  
  status() {}  
}
```

3. We will use this service in the `login` component. Before we get to the component itself, we need to support our component with forms. To support our login features via a form, import the supporting classes into `app.module.ts`:

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

4. Add this module to the imports array:

```
imports: [  
  BrowserModule,  
  AppRoutingModule,  
  ReactiveFormsModule  
]
```

You can close the `app.module.ts` file for now.

5. Now in the login component, change the template to the following:

```
<p>login works!</p>  
<h2 class="pb-2">Login Form</h2>  
<form [formGroup]="frmLogin" (ngSubmit)="onSubmit()">  
  <div class="form-group">  
    <label for="username">User name</label>  
    <input type="text" class="form-control" id="username"  
    formControlName="username">  
  </div>  
  <div class="form-group">  
    <label for="password">Password</label>  
    <input type="password" class="form-control" id="password"  
    formControlName="password">  
  </div>  
  <button type="submit" class="btn btn-primary">Submit</button>  
</form>
```

This is the same form from another boot camp. Once the Submit button is pressed, the `onSubmit()` function in the TS file is invoked. Remember to keep the `<router-outlet>` tags.

6. Now in the `login` component TS file, remove all the previously coded properties and remove all code from the `ngOnInit()` method. We now import the modules for supporting forms:

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

You can also remove the `employeeData` import at the top of that file. Also remove the injected references from the constructor.

7. Create a `FormGroup` type. The `!` is to prevent TS complaining about initialization:

```
export class LoginComponent implements OnInit {  
  frmLogin! : FormGroup;  
  constructor( ) {  
  }  
  ngOnInit(): void {  
  }  
}
```

8. If you had any code in `ngOnInit()` remove it. Add the function to create a `FormGroup` which will be used to assign to `frmLogin`:

```
ngOnInit(): void {  
}  
createFormGroup() {  
  return new FormGroup({  
    username : new FormControl(''),  
    password : new FormControl('')  
  })  
};
```

9. In the constructor, connect the `frmLogin` to the actual `FormGroup` object:

```
frmLogin! : FormGroup;  
constructor( ) {  
  this.frmLogin = this.createFormGroup();  
}
```

To prevent compilation errors, you can add an empty `onSubmit()` function.

PART 02 – IMPLEMENT PROFILE SERVICE

To use a service, build a class and implement the `@Injectable` method. If you provide the service in root, then the entire application can use the service. We import and inject the service into the component we wish to use. This creates an object in the background based on that service class. Once this happens, all the properties and methods that are public in the service will become available to the component using the service.

1. In the `login.component.ts` file, import the `profile.service.ts` file so that TS knows where the file is:

```
import { FormGroup, FormControl } from "@angular/forms";  
import { ProfileService } from "../../profile.service";  
@Component({
```

2. The service can now be injected into the class via the constructor:

```
export class LoginComponent implements OnInit {  
  frmLogin! : FormGroup;  
  constructor( private pService : ProfileService ) {  
    this.frmLogin = this.createFormGroup();  
  }  
}
```

3. Then in the `onSubmit()` method, we can utilize this service, specifically the login feature of that service:

```
onSubmit():void {  
  this.pService.login();  
};
```

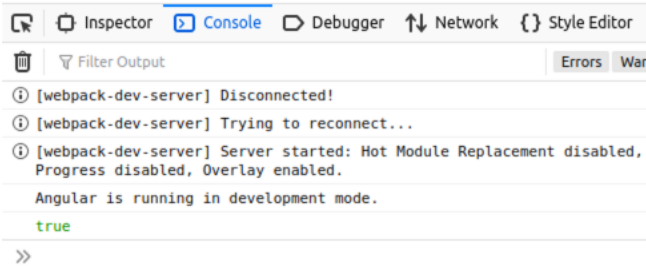
4. Log the result of this service call to make sure that it is working properly:

```
onSubmit():void {  
    console.log(this.pService.login());  
};
```

Login Form

User name

Password



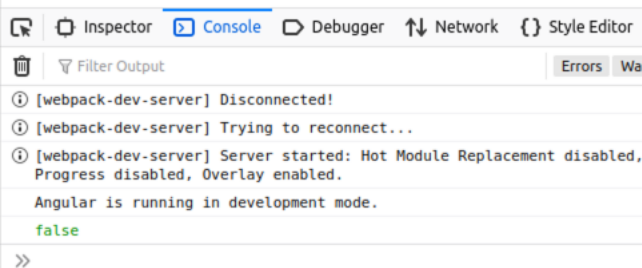
5. Now if you change the service to return false:

```
export class ProfileService {  
    login() {  
        return false;  
    }  
    logout() {
```

Login Form

User name

Password



PART 03 – ROUTE GUARDS

Route guards are used to restrict access to certain paths. It is NOT secure but can be used for basic authentication. Prior to Angular 14 classes were used to create guards. However recently Angular has moved to functions for this purpose.

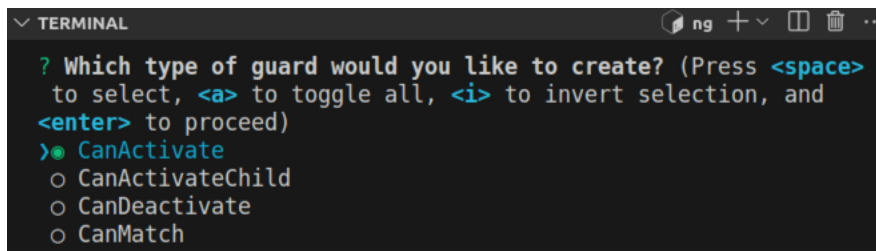
In this small example, we will restrict users from going into the *admin* part of our application. We will create a `canActivate` guard, then apply it to our path.

If you do not yet have an admin module, create one now using `ng g c admin`

1. In the terminal window, run the following command to create a guard function:

```
ng g g restricted --skip-tests
```

You will then be presented with a choice, just hit the <Enter> key to select `CanActivate`:



```
▼ TERMINAL
? Which type of guard would you like to create? (Press <space> to select, <a> to toggle all, <i> to invert selection, and <enter> to proceed)
> ● CanActivate
  ○ CanActivateChild
  ○ CanDeactivate
  ○ CanMatch
```

You should now have a file named `restricted.guard.ts`

2. The guard can now be configured with logic to either allow or restrict access. Our guard is simple for now, we allow everything by returning `true`:

```
import { CanActivateFn } from '@angular/router';
export const restrictedGuard: CanActivateFn = (route, state) => {
  return true;
};
```

3. To activate this guard, we import it into our `routes` array. Now just add the `canActivate` key and the value will be `restrictedGuard`:

```
{ path: 'login', component: LoginComponent, children: [
  { path: ':id', component: AdminComponent, canActivate: [restrictedGuard] },
],
{ path: '**', component: NotFoundComponent},
];
```

At this point, if `restrictedGuard` is not imported, just import it manually.

4. Spin the app and click on the *login Axle* link from the home page, we can see the *admin works!* Text, meaning that we can navigate to this child component. We added this admin component on Day01 Part 08.
5. Now if you change the return code of the `CanActivateFn()` function, you will not be able to see or navigate to the `admin` component:

```
export const restrictedGuard: CanActivateFn = (route, state) => {  
  return false;  
};
```

Remember the `CanActivateFn()` is in the `restricted.guard.ts` file. Although access is now restricted to the `admin` component, you could still navigate to the `login` view using the login menu link at the top of the view or the Login button on the home view.

PART 04 – INJECTING PROFILE SERVICE

It is becoming popular to use the `inject()` method to create instances of services in consuming classes and components. The result is the same but with this method we can inject services into functions also. Remember functions do not have constructors. Also, code inside of a constructor is executed as soon as the class gets called. So, with the injection method, we achieve a form of lazy injection.

1. In the `profile.service.ts` file, add a property to hold the status of our user's login:

```
export class ProfileService {  
  loginStatus : Boolean = false;  
  login() {  
    return false;  
  }  
  logout() {
```

2. Now use `loginStatus` to determine the state of login of the user:

```
export class ProfileService {  
  loginStatus : Boolean = false;  
  login() {  
    this.loginStatus = true;  
    return false;  
  }  
  logout() {  
    this.loginStatus = false;  
    return false;  
  }  
  status() {  
    return this.loginStatus;  
  }  
}
```

3. Then in the `restricted.guard.ts` file, import the `ProfileService`:

```
import { CanActivateFn } from '@angular/router';
import { ProfileService } from "../profile.service";
```

4. Once imported, we can use the `inject()` method to create an instance of this service:

```
export const restrictedGuard: CanActivateFn = (route, state) => {
  const profileService = inject(ProfileService);
  return false;
};
```

Remember to import the `inject` package from `@angular/core`.

5. Now change the `CanActivateFn()` function to return *true* or *false* based on what the `profileService` returns. So return whatever status the `profileService` contains:

```
export const restrictedGuard: CanActivateFn = (route, state) => {
  const profileService = inject(ProfileService);
  if(profileService.status() == true)
    return true;
  else
    return false;
};
```

Note, we just cant return `profileService.status()` . As it is now, this particular function accepts a type of `Boolean` whereas the `profileService.status()` functions returns a `boolean`. Another way to accept the value from `profileService.status()` is to use generics:
`return <boolean>profileService.status();`

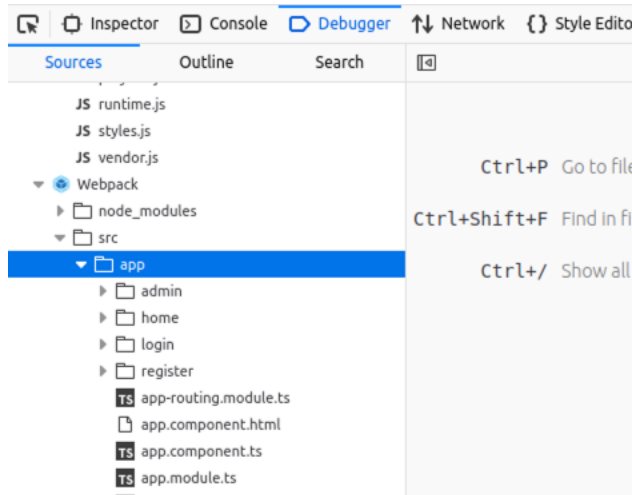
6. Back in the `profileService`, change the three methods to return the `loginStatus` only:

```
login(){
  this.loginStatus = true;
  return this.loginStatus;
}
logout(){
  this.loginStatus = false;
  return this.loginStatus;
}
status(){
  return this.loginStatus;
}
```

7. Test exactly like you did for Part03, it should work the same.

PART 05 – LAZY LOADING MODULES

Remove the restriction from `app.routing.module.ts` that prevents the user from going to that component.



Before proceeding, open the Debugger tab in Mozilla Firefox and drill down into the **Webpack** bundle. Sources tab in Google Chrome browser.

This bundle shows all the components, modules, CSS and other supporting files that will get loaded into memory at the start of the app.

In this bundle there is the *admin* folder. Even if we never use the admin components, it gets loaded into memory

and therefore uses resources. Lazy loading helps with this scenario. We only load components once they are needed. After implementing lazy loading of this admin component we will compare a second image of the resulting Webpack bundle.

1. In the terminal run this command. It will create two new files in the `admin` folder:

```
ng g m admin --routing
```

This would create two new files in the admin folder. Pay attention to the naming of these two files, one is `admin-routing.module.ts` file and the other is `admin.module.ts` file. Nowhere does it say *component*.

2. After completing #1, in the `admin.module.ts` file the `AdminRoutingModule` will be imported and made part of the `imports` array. So now in this file, import the component we want to lazy load, so `admin` component:

```
import { AdminComponent } from './admin.component';
@NgModule({
  declarations: [
    AdminComponent
  ],
  imports: [
```

Do this in `admin.module.ts`

3. If we run this code now we don't get an error but the `admin` component is being loaded into two different modules. To fix this, we should first comment or remove the existing path to `admin` from the `Routes` array in `app-routing.module.ts` file:

```
{ path: 'register', component: RegisterComponent },
{ path: 'login', component: LoginComponent, children: [
  { path: ':id', component: AdminComponent },
]},
{ path: '**', component: NotFoundComponent },
];
```

4. Now we use an Angular function called `loadChildren`. This function must point to an arrow function that returns the *module* you want to lazy load:

```
{ path: 'login', component: LoginComponent, children: [
  { path: ':id', loadChildren: () => import('./admin/admin.module')
    .then(m => m.AdminModule) },
]},
{
```

So, we replaced the component with a function that returns a module which will in turn load the component finally.

5. Remove the `AdminComponent` from `app.module.ts`. We can use a component in only one module:

```
import { RegisterComponent } from './register/register.component';
//import { AdminComponent } from './admin/admin.component';
@NgModule({
  declarations: [
    AppComponent,
    ...
    RegisterComponent,
    //AdminComponent,
  ],
  imports: [
```

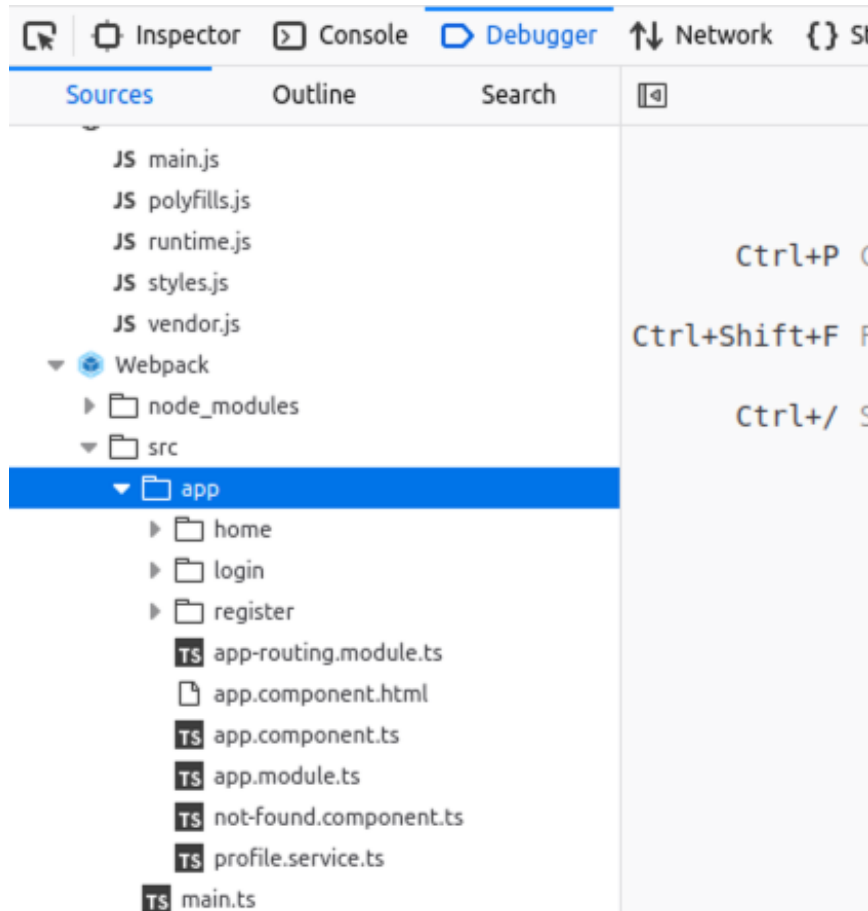
6. The final piece to this lazy loading puzzle is to create a new route for our admin component inside of `admin-routing.module.ts` file:

```
import { RouterModule, Routes } from '@angular/router';
import { AdminComponent } from './admin.component';
const routes: Routes = [
  {path: '', component: AdminComponent}
];
@NgModule({
```

7. In order to test the functionality, in Firefox open the Debugger tab in the Developers console window. Expand the *Webpack* area until you see *app*.

The Debugger is showing all the files and modules that were loaded. As you can see, the `admin` component is NOT loaded.

Even if you click on the login button or on any of the other two logins the admin module is NOT loaded. However, if you click on `loginAxle`, the module is loaded and shown in the Webpack bundle.



PART 06 – PASSING DATA AND RESOLVERS

On Day01 Part05 we saw that we can pass parameters into the TS code. In this section we see how to pass static data via the *routing* system.

1. In `app-routing.module.ts` file, add a data property to be passed into the admin component once it is loaded:

```
{ path: 'register', component: RegisterComponent },
{ path: 'login', component: LoginComponent, children: [
  { path: ':id',
    loadChildren: () =>
      import('./admin/admin.module')
      .then(m => m.AdminModule),
    data: { message: "Remember to logout" }
  },
] },
```

The name/key must be data. Note that this name/value pair, is NOT part of the `then()` method, it is a second statement after the `import()` method.

2. Then in the component, so `admin.component.ts` file, import `ActivatedRoute` from `Router` and inject that service into the component:

```
import { ActivatedRoute } from "@angular/router";
@Component({
  selector: 'app-admin',
  templateUrl: './admin.component.html',
  styleUrls: ['./admin.component.css']
})
export class AdminComponent implements OnInit {
  constructor(private activatedRoute : ActivatedRoute) { }
  ngOnInit(): void {
```

3. The data object can now be accessed via this `ActivatedRoute` feature:

```
export class AdminComponent implements OnInit {
  message : string = "";
  constructor(private activatedRoute : ActivatedRoute) {
    this.message = this.activatedRoute.snapshot.data["message"].toString();
    //console.log(this.message);
  }
}
```

As you can see, I created a property to hold the message. Then I passed the `activatedRoute` object to the property in the constructor. Notice that I had to extract the message key and turn it into a string. At the same time logged it to the console to make sure I am getting the proper data. Once you confirm everything you can remove the `log()` line.

4. To confirm the code so far works, click on the `login` `Axle` link on the `home` page/view. The message should show in the console window.
5. Now we extend this knowledge to the use of a `resolver`. A `resolver` will perform some action, usually a long running process and report back to the code that is using it. If everything is ok, the rest of the code continues to run. Let us create a `resolver`:

```
ng g r administrators --skip-tests;
```

A `resolver` is just a TypeScript file with one exported `const` value that resolves to `true` or `false`. `Resolvers` are considered as `middleware`. They are used to pre-load data before navigating to the page/view that needs that data.

6. The `resolver` is used in the `app-routing.module.ts` file:

```
path: 'admin',
loadChildren: () => import('./admin/admin.module')
  .then(m => m.AdminModule),
data: {message: "Remember to logout"},
resolve: {
  isAdminResolved: administratorsResolver
},
```

You will need to import `administratorsResolver` file.

7. To test the `resolver`, we need to change a few things in the `admin` component:

```
export class AdminComponent implements OnInit {
  message! : {};
  constructor(private activatedRoute : ActivatedRoute) {
    this.message = this.activatedRoute.snapshot.data;
    console.log(this.message);
  }
}
```

We change the `message` property to an object, then remove the `toString()` method from the `data` key we accessed before. In other words, we simply print the object itself rather than the properties of that object.

```
Angular is running in development mode.
```

```
► Object { message: "Remember to logout",
  isAdminResolved: true }
```

PART 07 – REALISTIC USE OF RESOLVERS

If you wish to go through this part, your JSON Server must be operational. In Appendix A there are instructions to install this server and start it up. This part is advanced. It involves use of the RxJS library and concepts of working with Observables.

1. We would need Angular's *http* module, so in the parent `app.module.ts` file import this module

```
import { AppComponent } from './app.component';
import { HomeComponent } from './home/home.component';
import { RegisterComponent } from './register/register.component';
import { HttpClientModule } from '@angular/common/http';
```

2. Add this module to the imports section:

```
],
imports: [
  BrowserModule,
  AppRoutingModule,
  HttpClientModule
],
providers: [],
```

(remember to insert a comma at the line above)

3. Although it is possible to perform the HTTP request inside of the `ResolveFn` function, it is much better to have the data return from a service:

```
ng g s administrators --skip-tests
```

4. Import the `HttpClient` module, then add a constructor to the service and pass in (inject) the `HttpClient`

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
@Injectable({
  providedIn: 'root'
})
export class AdministratorsService {
  constructor(private http: HttpClient) { }
}
```

This is the `administrators` service class. You get this class after running #3 above. Remember we also have the `administrators.resolver.ts` file.

5. For now, we will have just one single function that returns an `Observable`, so import the `Observable` module and then add this line:

```
constructor(private http: HttpClient) { }
getAdmins(): Observable<any>{ }
```

6. Then simply perform the `get` request to our json-server and return the data:

```
getAdmins(): Observable<any>{  
  return this.http.get("http://localhost:3000/employees");  
}
```

In effect, here we return an `observable` because `get()` from the `http` object, returns an `observable`.

7. Import the `AdministratorService` into the `administrators.resolver.ts` file:

```
import { ResolveFn } from '@angular/router';  
import { AdministratorsService } from "../administrators.service";  
import { inject } from '@angular/core';
```

Also import the `inject` module from `@angular/core`.

8. Configure the `resolver` class to work with the service. In the end we return the `observable` we got from the service:

```
export const administratorsResolver: ResolveFn<any> =  
(  
  route,  
  state,  
  administratorsService: AdministratorsService = inject(AdministratorsService)  
) => {  
  return true  
};
```

8. Now, instead of returning just `true`, return whatever `getAdmins()` returns to the `resolver`:

```
export const administratorsResolver: ResolveFn<any> =  
(  
  route,  
  state,  
  administratorsService: AdministratorsService = inject(AdministratorsService)  
) => {  
  return administratorsService.getAdmins();  
};
```

9. In `app.routing.module.ts` file, the code is mainly the same as in Part06 #5, 6 and 7. You may however, remove the message, it is not important now:

```
{ path: ':id',  
  loadChildren : () =>  
    import('../admin/admin.module')  
    .then(m => m.AdminModule),  
  data: {message: "Remember to logout"},  
  resolve: {  
    isAdminResolved: administratorsResolver  
  }  
},
```

10. The rest of the action takes place in the consumer file, so `admin.component.ts` and its template if you wish to go that far. Create an array to hold the data locked up in the `Observable` and comment out the message property. Also notice that we now need `ActivatedRoute`, so import it at the top of the file if it is not already there:

```
export class AdminComponent implements OnInit {  
  allAdmins : any = [];  
  //message! : {};  
  constructor(private activatedRoute : ActivatedRoute) {
```

11. The data we need is on the `data` property of the activated route, and accessible via snapshot.

```
export class AdminComponent implements OnInit {  
  allAdmins : any = [];  
  //message! : {};  
  constructor(private activatedRoute : ActivatedRoute) {  
    this.allAdmins = this.activatedRoute.snapshot.data;
```

12. We have to specifically add the `isAdminResolved` property as an array element of that data object. I added this property in Part 6 #6. It is around line 19 in the `app-routing.module.ts` file.

```
export class AdminComponent implements OnInit {  
  allAdmins : any = [];  
  //message! : {};  
  constructor(private activatedRoute : ActivatedRoute) {  
    this.allAdmins = this.activatedRoute.snapshot.data["isAdminResolved"];
```

13. If you want to see the result of all that code, continue to the template of the admin component and add these lines:

```
<p>admin works!</p>  
<ul class="list">  
  <li *ngFor="let admin of allAdmins">{{admin.username}}</li>  
</ul>
```

The screenshot shows a web application with a navigation bar at the top containing links for 'home', 'register', and 'login'. Below the navigation bar, there is a section titled 'login works!' followed by a 'Login Form'. The form has two input fields: 'User name' and 'Password', and a blue 'Submit' button. Below the form, there is another section titled 'admin works!' which displays a bulleted list of admin users: 'Axle', 'Jane', and 'Mary'.

The idea here is that Resolver resolved getting a list of admins from a data source. We then wrote code to access that list and show it on the browser window.

Angular 16 has introduced *automatic route parameter mapping* using the `@Input()` decorator. It eliminates the need for an `ActivatedRoute` service. In the end we write less code but have the same functionality.

1. We used the `@Input()` functionality in other boot camps. In this simple example, the parameter that carries the array of administrators is finally utilized in the `AdminComponent`. We already have an `allAdmins` property to hold the resulting array from the `Administrator` service. So, just declare the decorator there for now:

```
styleUrls: ['./admin.component.css']
}))
export class AdminComponent {
  allAdmins : any = [];
  @Input()
```

If the `Input` module is not automatically imported, do it manually. Remember that if `isAdminResolved` is true, we get an array of `Administrators`.

2. This `AdminComponent` is the final call for the path or route that is triggered. This means that everything on that route is accessible to this component. This includes the `isAdminResolved` property. Recall that this property was set in the `app-routing.module.ts` file. This property, which is part of the resolver property, collects data from our database after the resolver is resolved. We can therefore use that property here by pointing our `@Input()` decorator function to it:

```
styleUrls: ['./admin.component.css']
}))
export class AdminComponent {
  allAdmins : any = [];
  @Input() isAdminResolved : [] = [];
```

3. Since we do not need the constructor anymore, we can remove it. However we need to pass our data to something. This will now be done in the `ngOnInit()` method. So import the `OnInit` module from `@angular/core` and implement the `OnInit()` method:

```
export class AdminComponent implements OnInit{
  allAdmins : any = [];
  @Input() isAdminResolved : [] = [];
```

We also do not need the `ActivatedRoute` module from `@angular/router`

- 4.

5. Finally for this file, we can access our data once resolved, using `ngOnInit()` :

```
@Input() isAdminResolved :[] = [];  
ngOnInit(): void {  
    this.allAdmins = this.isAdminResolved;  
}
```

6. Since this is a breaking change for version 16, the changes go deep. We now have to instruct the `app-routing` module that we will be using *automatic route parameters*:

```
{ path: '**', component:NotFoundComponent},  
];  
@NgModule({  
    imports: [RouterModule.forRoot(routes,{bindToComponentInputs:true})],  
    exports: [RouterModule]  
})  
export class AppRoutingModule { }
```

This change is in the `app-routing.module.ts` file.

7. No other changes are necessary, you may test and find that you get the same result as in Part07 #13

home register login

login works!

Login Form

User name

Password

Submit

admin works!

- Axle
- Jane
- Mary

APPENDIX A – JSON SERVER

If you were part of the Angular 16 Intro boot camp, you would remember installing the JSON Server. This is the same set of instructions from that BC. We would use the JSON server to create a service and handle our users that way.

1. We would need a mock server so that we can make API calls. Install the JSON Server using the command: `npm install json-server --save-dev`
Make sure that you are in the skills folder when you do this.
2. This server will need a database file to work with. Create a data folder inside of your app folder. Now move your db.json file into that data folder. First we need to configure our package.json file to run our server. In package.json file, go to the *scripts* section and add a new script as shown below:

```
"version": "0.0.0",
"scripts": {
  "ng": "ng",
  "start": "ng serve",
  "build": "ng build",
  "test": "ng test",
  "lint": "ng lint",
  "e2e": "ng e2e",
  "server": "json-server --watch ./src/app/data/db.json"
},
"private": true,
```

Remember to insert a comma at the line above. The server is watching a file called db.json You used this file on Day01 Part 05. It is also available in your downloads from GitHub. You could drag existing db file into the data folder.

3. Start the server by going back to a terminal window and run the command:
`npm run Server`

```
95 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities

axle@pc0481:~/Documents/Skills$ npm run Server
> skills@0.0.0 Server /home/axle/Documents/Skills
> json-server --watch ./src/app/data/db.json

\{^_^\}/ ht!

Loading ./src/app/data/db.json
Done

Resources
http://localhost:3000/customers

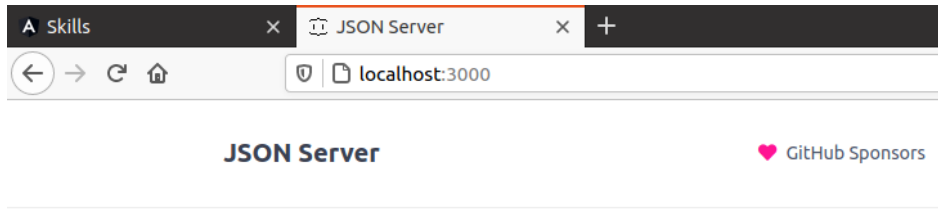
Home
http://localhost:3000

Type s + enter at any time to create a snapshot of the database
Watching...
```

The computer should respond with a message and a location of where it can be accessed with a browser, *localhost:3000* in this case. Better not to do this from VS Code terminal window.

4. Now go to that location using a

browser



Notice that under **Resources** the server found our *employees* database

Note to stop any process in Linux just hold down the CTRL button and then hit the letter C on the keyboard.