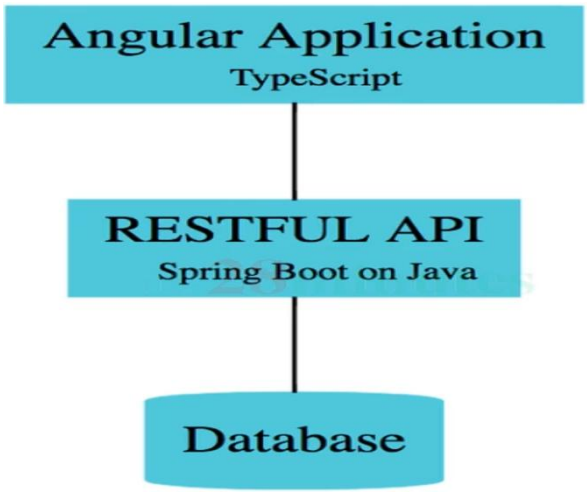
Angular

Planning:
ANGULAR FRONTEND (TYPESCRIPT->JAVASCRIPT+STRONG TYPING) -> SPRING BOOT REST API (JAVA) -> SECURITY AND AUTHENTICATION SETTINGS -> DATABASE.

Architecture:



BROWSER->ANGULAR APP->RESTFUL API->DATABASE -> RESTFUL API->ANUGLAR APP->ROWSER

The angular structure use the modern JavaScript, where you have the HTML,CSS and JavaScript files separated, but imported as a single file and the dependencies are known as modules.

import {Component} from “@angular/core”<-module;

ES7
@Component({
selector: ‘app-root’->tag that will say where the component will be rended on index.html=> <html><app-root></html>
templateUrl: ‘./app.component.html’;
styleUrls: [‘./app.component.css’]
})
export class AppComponent {
title = ‘todo’;-> property / atribute;
}

JAVASCRIPT VERSION

ES = EcmaScript

- ES3 - 1999
- ES5 - 2009
- ES6 - 2015 - ES2015
- ES7 - 2016 - ES2016
- ES8 - 2017 - ES2017

EcmaScript is the pattern established like an interface and the javascript is the implementation.

JavaScript x JavaxTypeScript

JS-> No types
var value; (undefined type, can receive any type)
value = 5 (number)
value = “five” (text)

```
function add (number1,number2) {
        return number1+number2;
}

interface DoSomething{}
```

JAVA-> With types
int value; (defined type to receive specific type of value)
value = 5 (number)\/
value = “five” (text)XXX Compilation error

```
public int add (int number1, int number2) {
        return number1+number2;
}
public interface DoSomething{}
```

TypeScript
value:number; (defined type to receive specific type of value)
value = 1 (number)\/
value = “five” (text)XXX Compilation error

```
function add (number1: number,number2:number):number {
        return number1+number2;
}
```

interface DoSomething{}

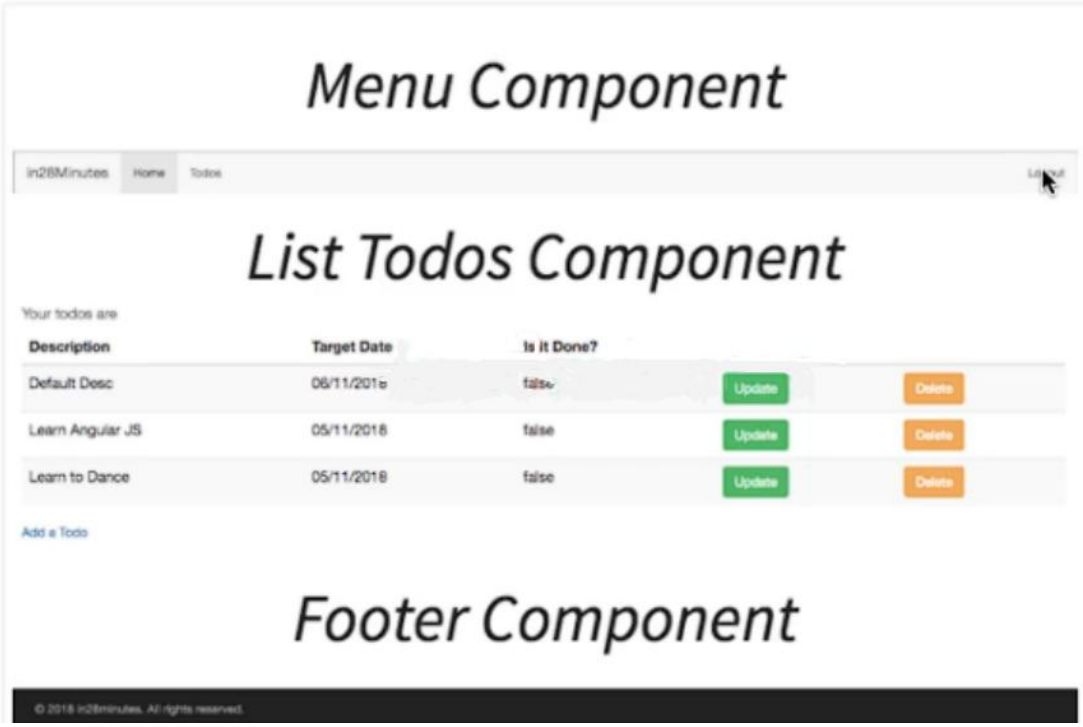
For creating and managing the angular projects we use the command line interface from Angular (angular cli), installing the node packages through https://nodejs.org/en/download, then npm install -g @angular/cli.

Once installed, we have the below commands:

- ✓ **ng new** project name -> it creates the whole structure for an angular project;
- ✓ **ng lint (sonar)** -> it checks if the code is following the pattern of clean code;
- ✓ **ng build->** it generates an '.apk,.exe' of the project that is compiled to run in an external server with all the files html, css and where .ts was translated to .js;
- ✓ **ng dist ->** once built, it runs the app on prod;
- ✓ **ng test->** it generates and run the test scenarios with the name of specifications to point success or failure;
- ✓ **ng e2e->** it builds the app, test it and run it to confirm if there are errors in the process
- ✓ **ng generate->** it creates new components,routes, configs and services
- ✓ **ng serve->** it runs the application locally.

The directory app with all the components needed and the routes, in the root, assets for images,videos,sounds, environments for local or prod env. We have the package.json where we define the modules that will be used, the main.ts where we initialize the application telling we want to up the content from appComponent as the root component, the index.html that is the page that will be first loaded with a base html page and the content of the root component, also we have the styles.css file containing the global css for the app.

COMPONENTS



Components represent parts of the page that will return the template (html), the style (css) and the interaction (typescript) when clicking in some button for example update a to do, delete a todo, add a todo.

```
2
3  @Component({
4    selector: 'app-root',
5    templateUrl: './app.component.html',
6    styleUrls: ['./app.component.css']
7  })
8  export class AppComponent {
9    title = 'todo';
0  }
1
```

@Component is the decorator like an annotation that identifies the class as a component that will import the html and css content.

As parameters, we pass the selector that will represent the tag to be render in the index.html with whole content comprising the template and the style .

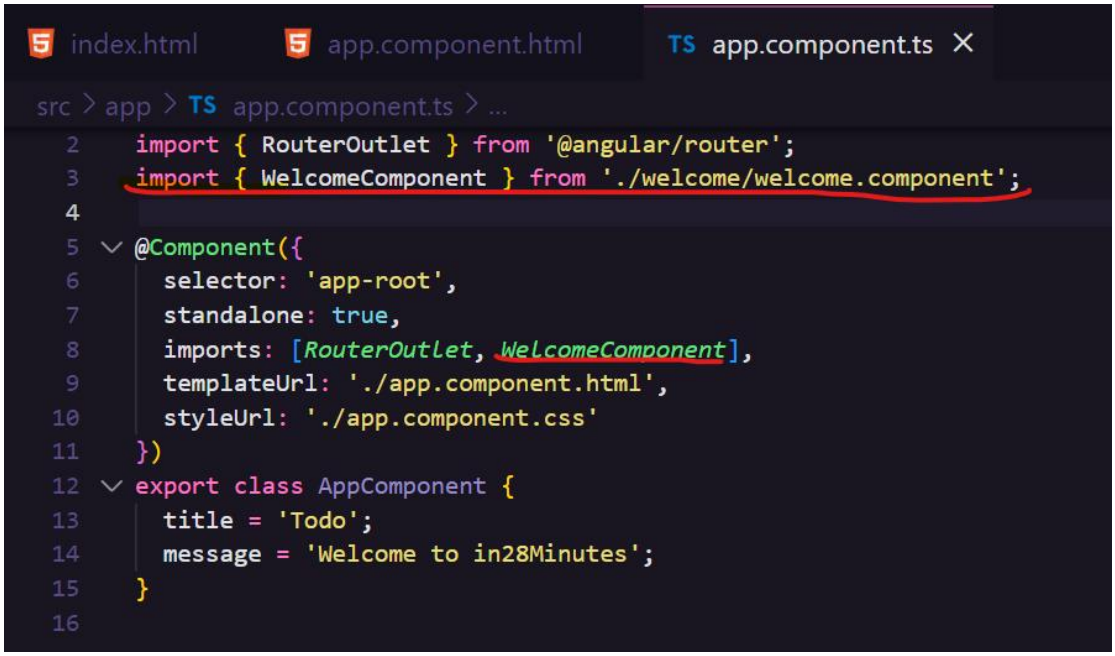
```
1
2  <div style="text-align:center">
3    <h1>
4      Welcome to {{ title }}!
5    </h1>
6    <img width="300" alt="Angular Logo" src="data:image/s
7  </div>
8
9  <router-outlet></router-outlet>
10
```

The property ‘title’ is defined in the typescript class and its value can be used in the template through the **interpolation {{**property**}}**, being part of data biding from component to template.

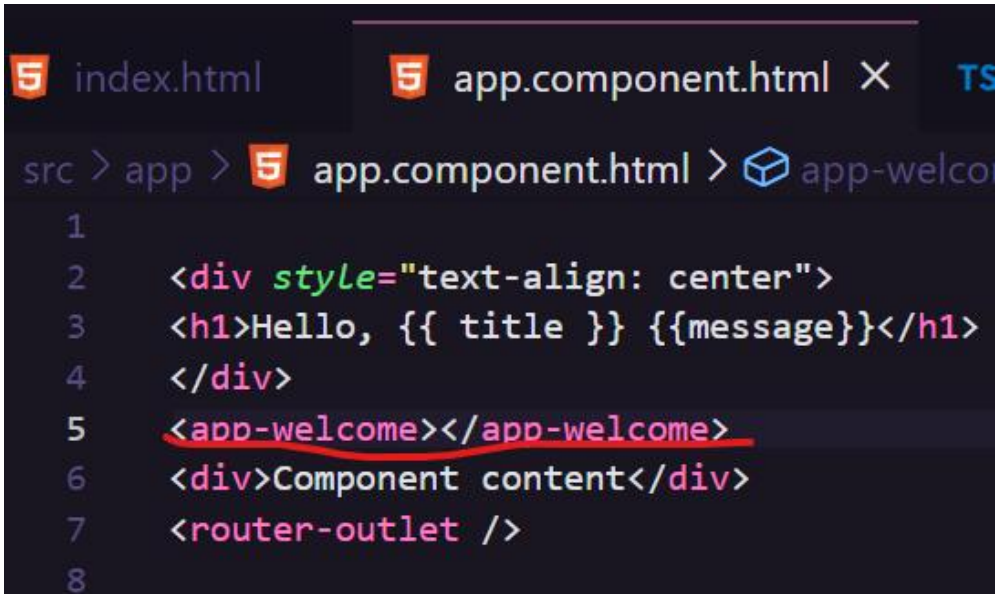
```
                                    ng generate component welcome
CREATE src/app/welcome/welcome.component.css (0 bytes)
CREATE src/app/welcome/welcome.component.html (26 bytes)
CREATE src/app/welcome/welcome.component.spec.ts (635 bytes)
CREATE src/app/welcome/welcome.component.ts (273 bytes)
```

To create a new component considering all the four files (html,css,spec.ts, ts) at once, we use the command ng generate component ‘name’ in a separate terminal.

So, to render this new component, we need to import it to the root component:



And then use its content in the root template through the tag:



Having one component inside another.

The reason why the new component doesn't need to be declared on @ngModule on app.module.ts to be used is that, since angular 13, it is independent and just need to be imported to be used in any place.

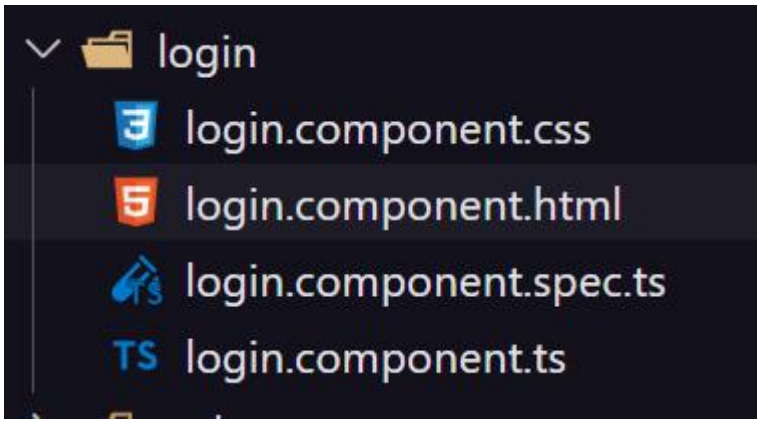
Standalone Update: Why Angular Standalone Components?

Angular 13 has introduced a novel functionality known as Angular standalone components. These components streamline the process of Angular development and minimize the need for repetitive code. Unlike conventional Angular modules, you DO NOT need to have NgModule files for standalone components. Consequently, you can effortlessly import and utilize them in any section of an application.

Benefits

- ✓ Improved Developer Experience
- ✓ Increased modularity
- ✓ Improved performance

More details here : https://github.com/in28minutes/full-stack-with-angular-and-spring-boot/blob/master/step-by-step-changes-to-standalone.md#why-angular-standalone-components



ng generate component login --skip-import --standalone

app.component.ts | login.component.html | login.component.css

src > app > login > login.component.html > div.form > div.items > fieldset

```html
<div class="logo">
    <img src="../../assets/ImageToDo.png" id="logo"/>
    <!--https://bartonfamilylaw.com.au/wp-content/uploads/2018/05/to-do.j
</div>
<div class="form">
    <div class="items">
        <fieldset>
            <label for="username">User Name:</label>
            <input type="text" id="username" name="username"><br>
            <label for="password">Password:</label>
            <input type="password" id="password" name="password"><br>
            <div class="button">
                <input type="submit" value="Enter" id="enter">
            </div>
        </fieldset>
    </div>
</div>
```

```css
.logo {
    display: flex;
    justify-content: center;
    align-items: center;
    margin-top: 50px;
}
.logo img#logo {
    height: 280px;
    width: 280px;
}
.form {
    display: flex;
    justify-content: center;
    align-items: center;
    min-height: calc(100vh - 625px);
}
```

```css
.items {
    display: flex;
    justify-content: center;
    align-items: center;
    background-color: #fff740;
    font-family: 'Gill Sans', 'Gill Sans MT', Calibri, 'Trebuchet MS', sans-serif;
    font-size: 22px;
}
```

```css
.items fieldset {
    border: 2px solid black;
}
```

```css
.items label,
input {
    margin: 20px;
}
```

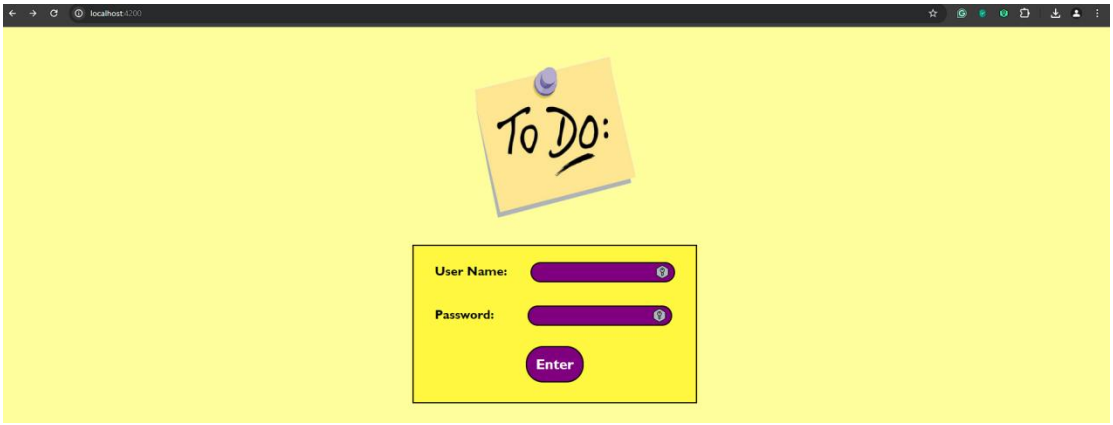
```css
.items label {
    color: black;
    font-weight: 700;
}
```

```css
.items input {
    color: white;
    font-weight: 700;
    background-color: purple;
    padding: 5px;
    border: 2px solid black;
    border-radius: 30px;
}
```

```css
.items input#password {
    margin-left: 38px;
}
```

```css
.button {
    display: flex;
    justify-content: center;
    align-items: center;
}
```

```css
.button input#enter {
    margin-top: 15px;
    background-color: purple;
    font-weight: 700;
    color: white;
    font-family: "Gill Sans", "Gill Sans MT", Calibri, "Trebuchet MS", sans-serif;
    font-size: 25px;
    padding: 15px;
    border: 2px solid black;
    border-radius: 30px;
    cursor: pointer;

}
```



Now once, we write the username and password, we want to make the login actually. For that happening, we declare the click event in the button between parenthesis (click) then we point what action should be thrown like below:

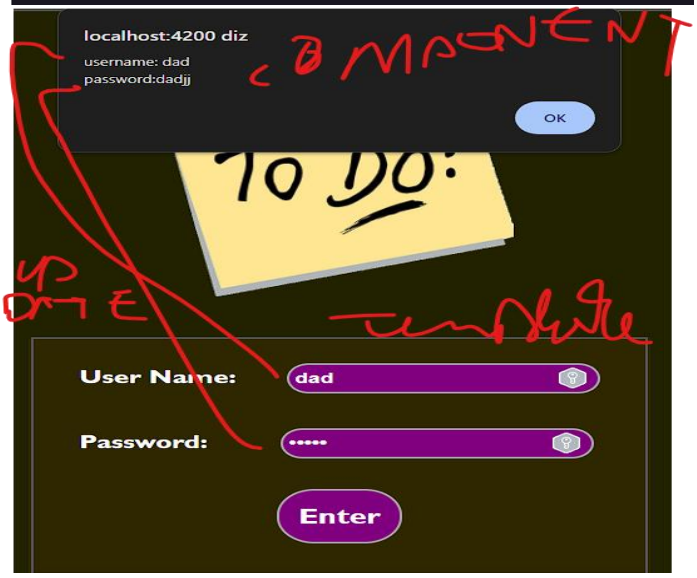
```
<div class="button">
    <input type="submit" value="Enter" id="enter" (click)="handleLogin()">
</div>
```

```
export class LoginComponent {

  username = 'in28Minutes';
  password = '';

  handleLogin(){
    alert(this.username);
  }
}
```

This thrown action is defined in the logic view (component), in the example we show the value of the property username.

```
<label for="username">User Name:</label>
<input [(ngModel)]="nameperson" type="text" id="username" name="username"><br>
<label for="password">Password:</label>
<input [(ngModel)]="password" type="password" id="password" name="password"><br>
```

```
import { Component } from '@angular/core';
import { FormsModule } from '@angular/forms';

@Component({
  selector: 'app-login',
  standalone: true,
  imports: [FormsModule],
  templateUrl: './login.component.html',
  styleUrl: './login.component.css'
})
export class LoginComponent {
  nameperson = 'in28Minutes';
  password = '';

  handleLogin(){
    alert("username: "+this.nameperson+ "\npassword:"+ this.password);
  }
}
```

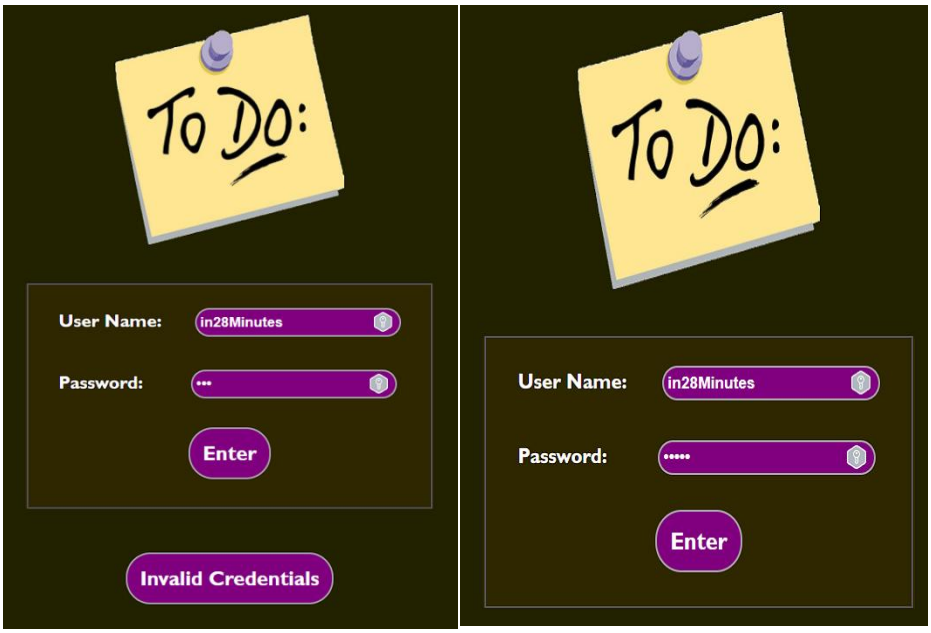


The banana in the box approach [()] is a mix of interpolation and event biding because inside the template, we recover the value of the component property and update its value when typing new values in the template through [(ngModel)]=”property” this is called two-way biding, as in, the data link between component and template, template and component.

```
errorMessage = 'Invalid Credentials';
invalidLogin = false;

handleLogin() {
  if (this.nameperson === "in28Minutes" && this.password === "dummy") {
    this.invalidLogin = false;
  } else {
    this.invalidLogin = true;
  }
}
```

For a simple authentication step for now, we can hardcore it, creating properties for showing that the login is invalid or not through a text and a state checking the username and password.

```
<small *ngIf='invalidLogin'>{{errorMessage}}</small>
```

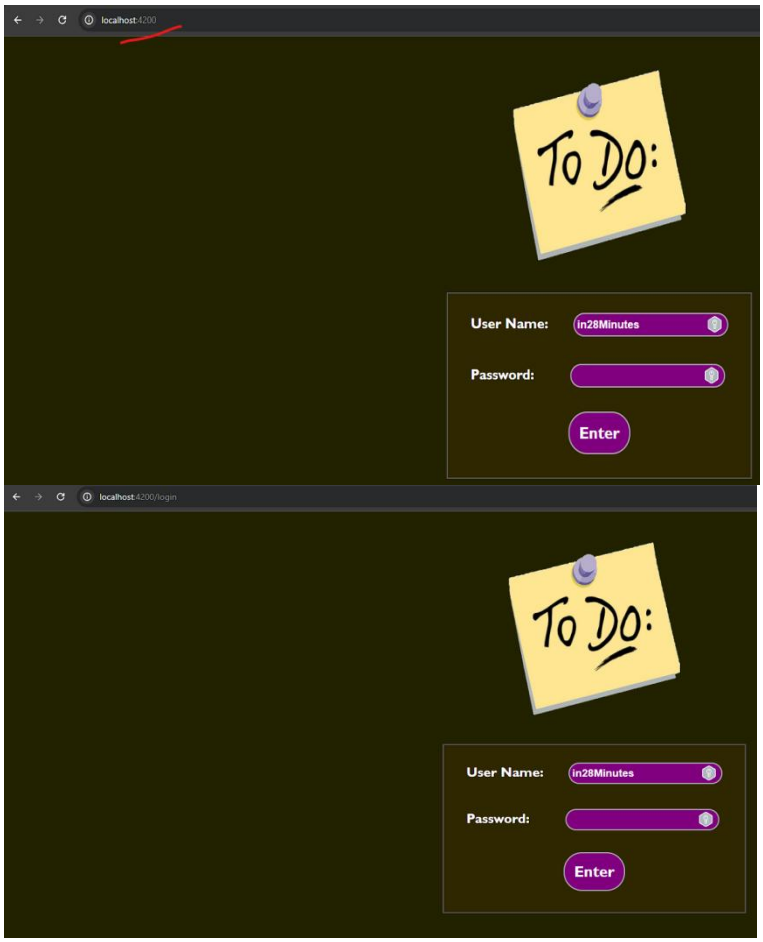


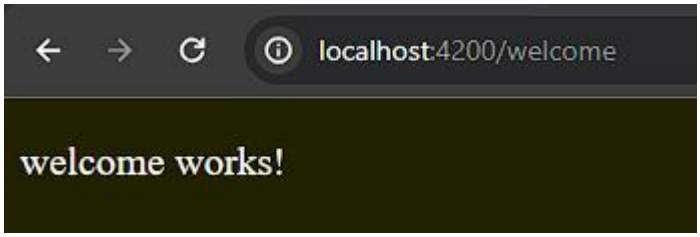
This is an opportunity for another directive (broaden the meaning), for showing the message or not in the template, we use *ngIf=”state”, as in, if the state is true the content is displayed, if the login is invalid show the message error.

```
TS login.component.ts   TS app.routes.ts ×   app.component
src > app > TS app.routes.ts > routes
1   import { Routes } from '@angular/router';
2   import { LoginComponent } from './login/login.component'
3   import { WelcomeComponent } from './welcome/welcome.comp
4   import { ErrorComponent } from './error/error.component'
5
6   export const routes: Routes = [
7       {path:'', component : LoginComponent},
8       {path:'login', component :  LoginComponent},
9       {path:'welcome', component : WelcomeComponent},
10      {path:'**', component : ErrorComponent}
11  ];
12
```

src > app > app.component.html > router-outlet

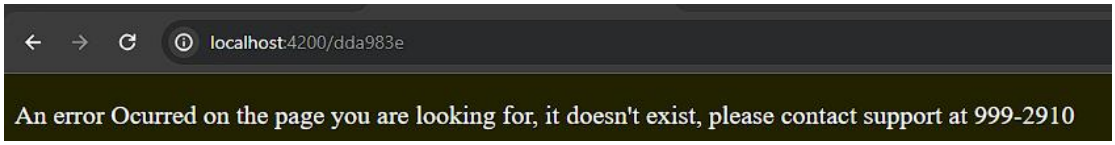
```
1   <!--<app-welcome></app-welcome>-->
2   <!--<app-login></app-login>-->
3   <!--<div>Component content</div>-->
4   <router-outlet></router-outlet>
5
```





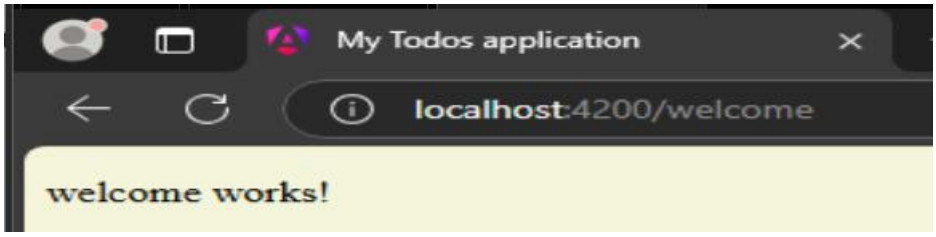
```ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-error',
  standalone: true,
  imports: [],
  templateUrl: './error.component.html',
  styleUrl: './error.component.css'
})
export class ErrorComponent {

  errorMessage = "An error Ocurred the page you are looking for doesn't exist, please contact support at 999-2910 "
}
```

```html
<p>{{errorMessage}}</p>
```



The next step is to 'route' our components, as in, map the possible routes(endpoints,urls) and render the specific component, in case any invalid route is typed (**) like a 404 page(not found) render the specific component.

```
export class LoginComponent {
  invalidLogin : boolean = false;

  // dependency injection (@Autowired)
  constructor(private router: Router){}

  handleLogin() : void{
    if(this.username === 'In28Minutes' && this.password === 'dummy'){
      //redirecting the user to the welcome component page
      this.router.navigate(['welcome'])
      this.invalidLogin = false;
    }else{
      this.invalidLogin = true;
    }

  }
}
```

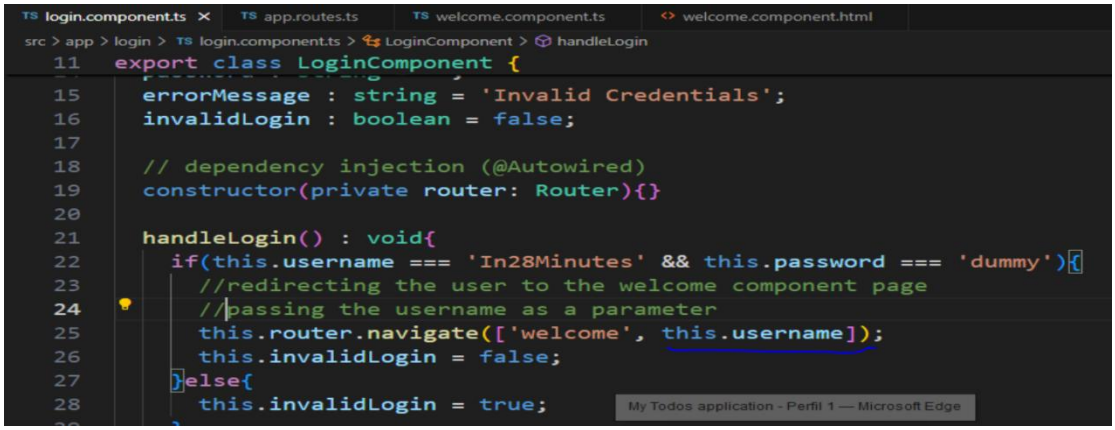


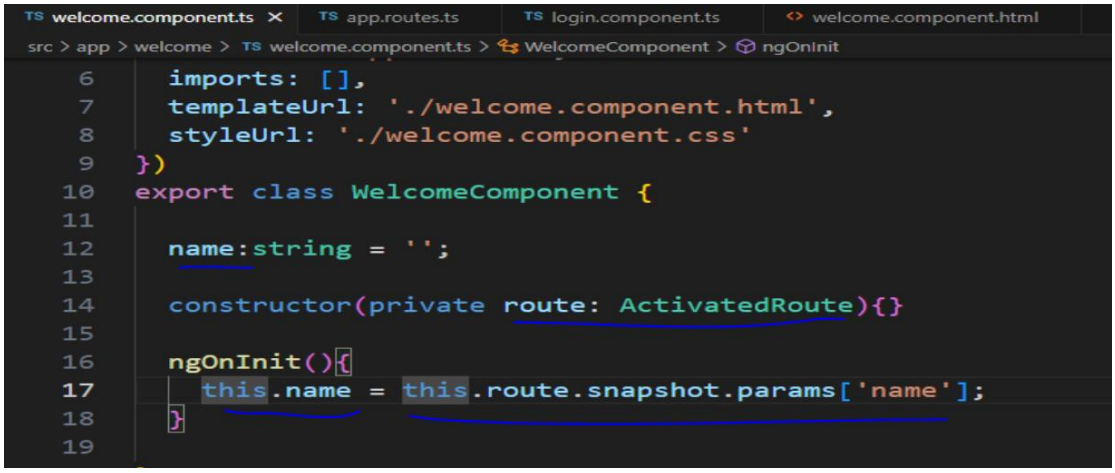
To make the login button redirect the user to the welcome page, in login typescript file is needed to include a dependency injection of the **router module** for then use it when the correct username and password be entered, the browser navigates to the welcome page (component).

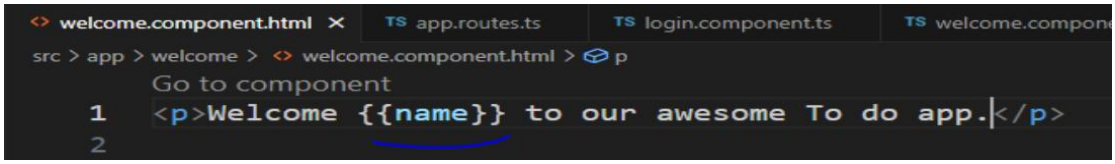
app.routes.ts | welcome.component.ts | welcome.component.html | login.component.ts

src > app > app.routes.ts > routes > path

```
1  import { Routes } from '@angular/router';
2  import { LoginComponent } from './login/login.component';
3  import { WelcomeComponent } from './welcome/welcome.compon
4  import { ErrorComponent } from './error/error.component';
5
6  //mapping the possible routes to render the correct compon
7  export const routes: Routes = [
8      {path: '', component: LoginComponent},
9      {path: 'login', component: LoginComponent},
10     {path: 'welcome/:name', component: WelcomeComponent},
11     {path: '**', component: ErrorComponent}
12 ];
```







On the welcome page, we want to identify the user that is logged in our application, for that, it is needed to declare on app.routes, the welcome path with the parameter -> **path: 'welcome/:name'**. On the login file, pass the parameter when redirecting to the welcome page -> **this.route.navigate(['welcome', this.username]).**

On the welcome file(.ts), capture its value using the constructor, through the dependency injection object of the type **ActivatedRoute,** set this value to a property to be shown in the template via interpolation -> **this. name = this.route.snapshot.params['name'] (.ts) {{name}} (.html).**

app.routes.ts | list-todos.component.ts 3 | list-todos.component.html

src > app > app.routes.ts > routes > component

```ts
import { Routes } from '@angular/router';
import { LoginComponent } from './login/login.component';
import { WelcomeComponent } from './welcome/welcome.compon
import { ErrorComponent } from './error/error.component';
import { ListTodosComponent } from './list-todos/list-todo

//mapping the possible routes to render the correct compon
export const routes: Routes = [
    {path: '', component: LoginComponent},
    {path: 'login', component: LoginComponent},
    {path: 'welcome/:name', component: WelcomeComponent},
    {path: 'todos', component: ListTodosComponent},
    {path: '**', component: ErrorComponent}
];
```

app.routes.ts | list-todos.component.ts 3 | list-todos.component.html

src > app > list-todos > list-todos.component.ts > ListTodosComponent > todos

```ts
import { Component } from '@angular/core';
import {NgIf,NgFor, UpperCasePipe,DatePipe} from '@angular
@Component({
  selector: 'app-list-todos',
  imports: [NgIf,NgFor, UpperCasePipe,DatePipe],
  templateUrl: './list-todos.component.html',
  styleUrl: './list-todos.component.css'
})
export class ListTodosComponent {

  todos = [
    {id: 1, description : 'Learn to Dance'},
    {id: 2, description : 'Become an Expert at Angular'},
    {id: 3, description : 'Visit India'}
  ];
}

```

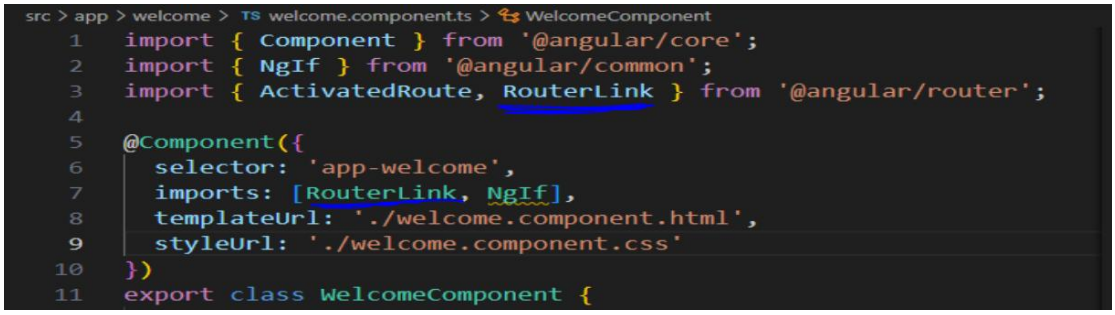
app.routes.ts | list-todos.component.ts 3 | list-todos.component.html

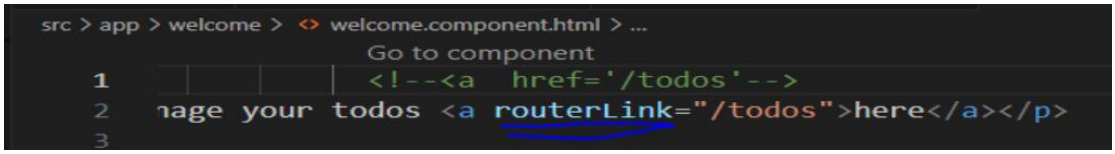
src > ap C:\Angular Projects\todo\src\app\app.routes.ts l > table > tbody

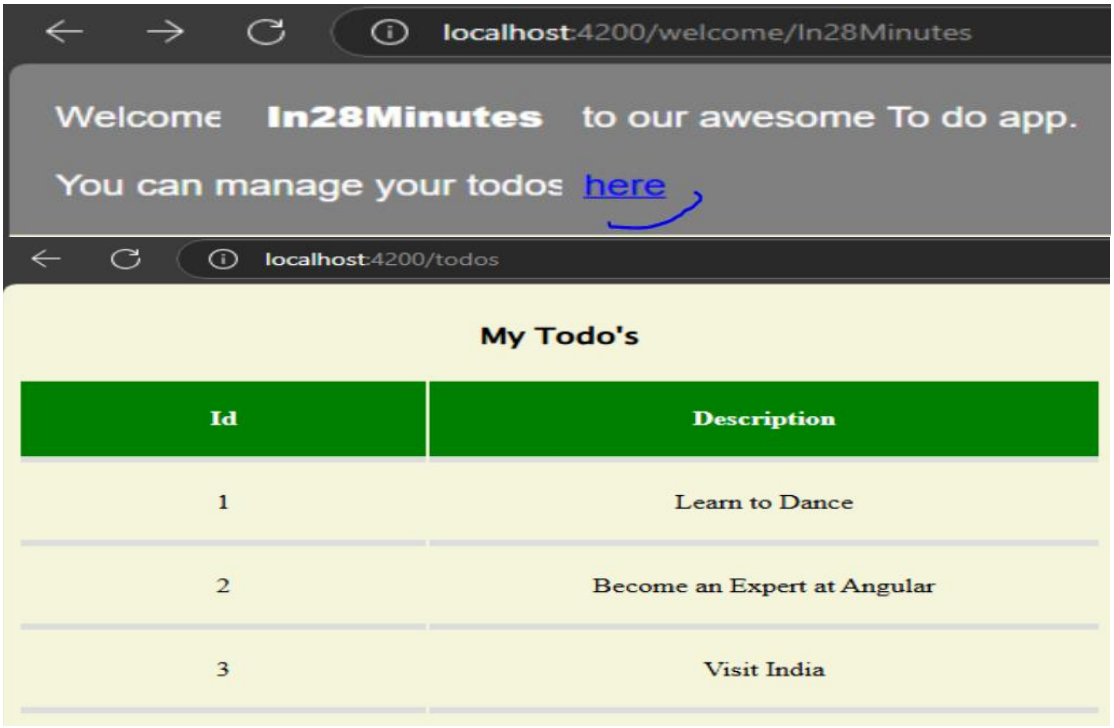
Go to component

```html
<table>
    <caption>My Todo's</caption>
    <thead>
        <tr>
            <th>Id</th>
            <th>Description</th>
        </tr>
    </thead>
    <tbody>
        <!--for (Todo todo : todos)
        (for each element todo inside todos)-->
        <tr *ngFor="let todo of todos">
            <td>{{todo.id}}</td>
            <td>{{todo.description}}</td>
        </tr>
    </tbody>
</table>

```

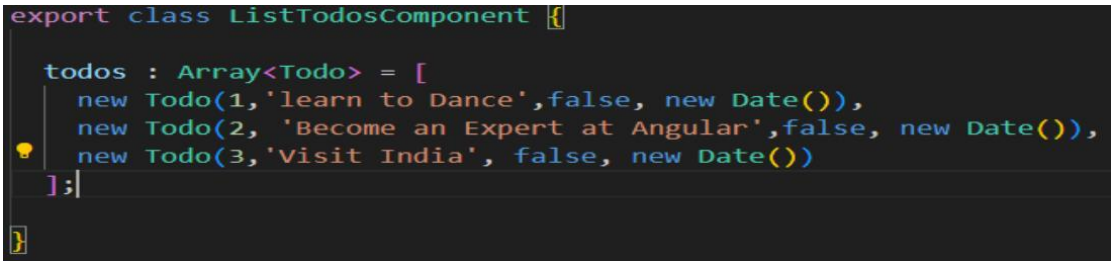
On the listTodo component, we want to create a list of todos. We map this component on app.routes.ts ->**{path: 'todos', component: listTodoComponent}.** Create a list of todos on the ts file -> **todos = [{id: 1, description: 'Learn to dance'},{id:2, description: 'Become an Expert at Angular'},{id: 3, description: 'Visit India'}].** On the template inside the table structure -> **<tr *ngFor=' let todo of todos'><td>{{todo.id}}</td><td>{{todo.description}}</td></tr>** , the directive *ngFor represents the repeating structure **for,** making looping a list, as in, sweeping the list saying for each element inside the list do an action.





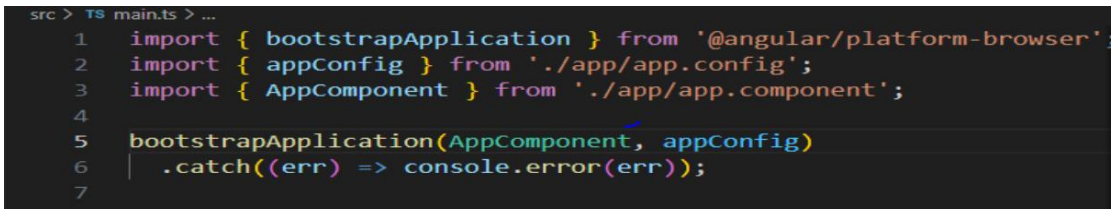


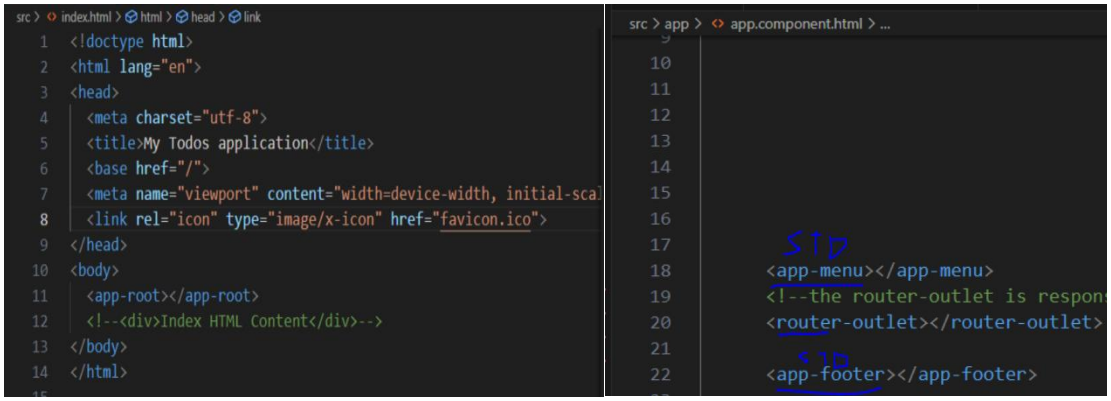
Another way to navigate between components is to create a link to connect them, to solve this, the **routerLink directive** is used inside the **<a>** tag as an attribute.



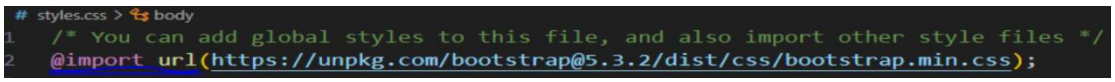
A more feasible way to control the todo's list, it is to create a todo class -> **export class Todo {constructor(public id: number, public description: string, public done:boolean, public targetDate: date)}**(todo.ts) to create objects / instances of this class -> **todo : Array<Todo> = [new Todo(1, 'learn to Dance', false, new Date()]**(todo.ts). Then, render the content on the template table -> **<tr *ngFor="let todo of todos"><td>{{todo.description}} </td><td>{{todo.targetDate | date | uppercase}}</td><td>{{todo.done}}</td> </tr>.**

Just a quick review about modules, any .ts file can be consider one module that will contains the logic or functionalities of the system with one or more classes in it and in angular all these files together make part of a group of components that will for form an angular module to be rendered on the screen (template), like this: @NgModule({ declarations: [AppComponent, WelcomeComponent,LoginComponent,ErrorComponent,ListTodosComponent].

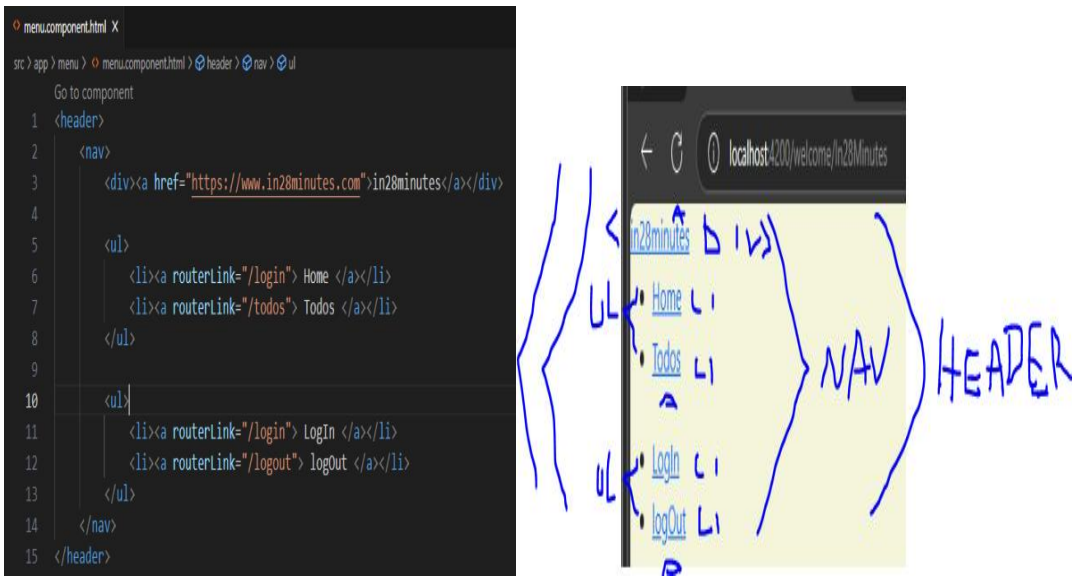




As a quick review, the appComponent is the root component, because it's always the first component to be rendered in the index.html (root template). It's because the **angular** bootstrap is used to initialize the appComponent as the root component along with the routes of the project and the standalone components to be rendered at the **index.html** (<app-root>).



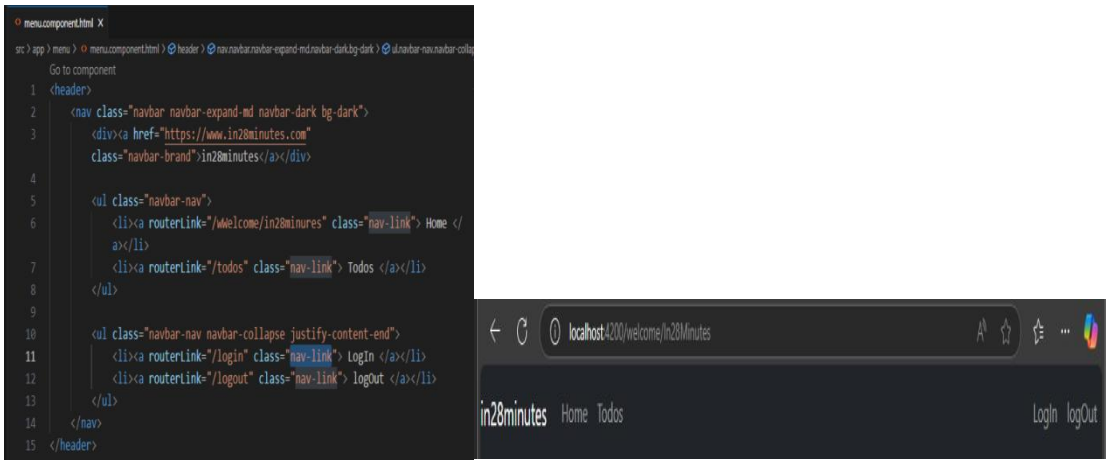
On the project, the bootstrap will be used as a framework to style the application in a more optimized way, instead of CSS styles codes directly.



Using the html structure and angular on template, the tags are:

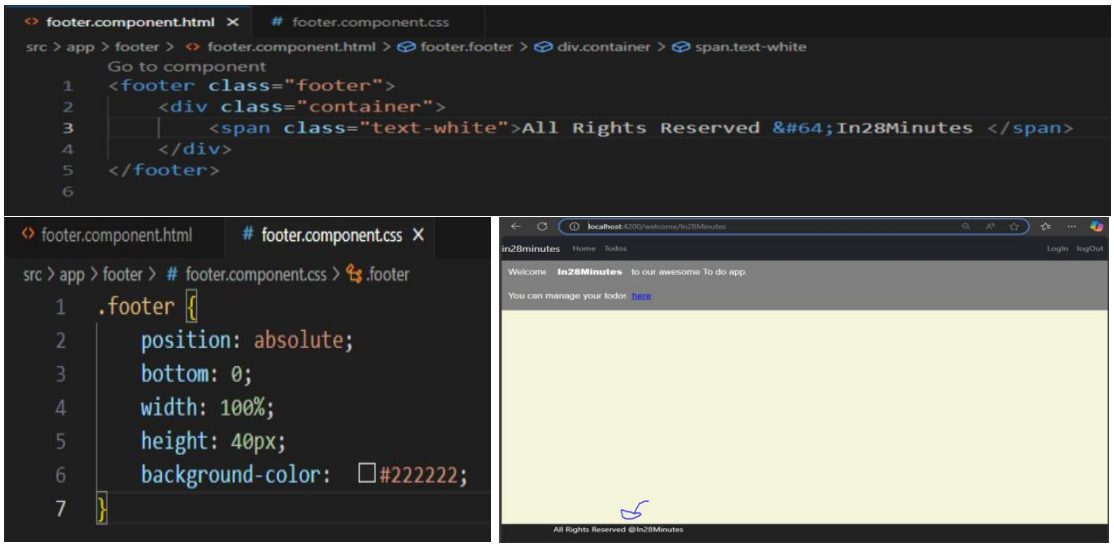
<header> means that everything in header, will be consider a header area </header>[1]
<nav>means that everything in nav, will be consider a navigation area</nav>[1]
 division area <a routerLink[2]="route"> with an anchor to redirect to another route

[1] semantic meaning, not visual
[2] angular directive to communicate template and logic view to replace the href attribute



Bootstrap css classes:

navbar -> style to identify the navigation bar
navbar-expand-md -> style to expand the navigation bar in medium size
navbar-dark bg-dark -> style to change the background color of the navigation bar to dark black
navbar-brand -> style to stand out the brand
navbar-nav-> style to identify the navigation bar and padding
navbar-collapse justify-content-end -> style to identify the navigation bar and align the items at the right corner ending
nav-link-> navigation link style



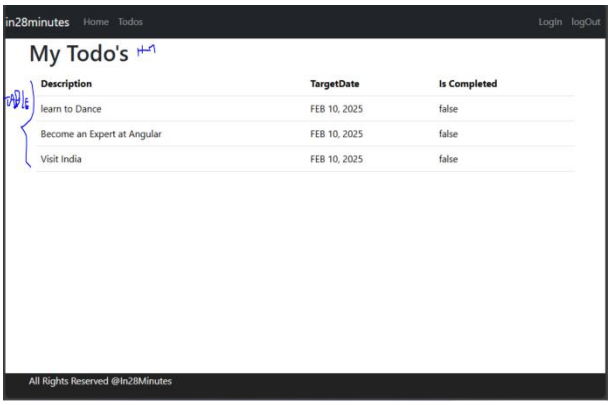
@ -> @ symbol
The tag <footer> means that content inside it will be the footer area.
The tag means a little piece of content with less importance

Bootstrap css classes:
container
text-white

```
src > app > app.component.html > div.container
11
12
13
14
15
16
17  <app-menu></app-menu>
18
19  <div class="container">
20      <!--the router-outlet is responsible to render the
        correct components according to the route-->
21      <router-outlet></router-outlet>
22  </div>
23  <app-footer></app-footer>
```

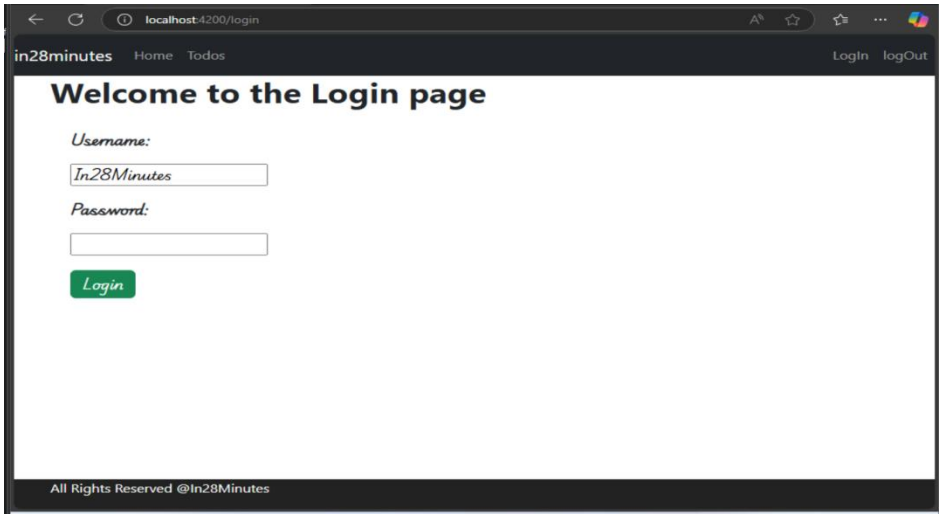
Class container used to give more padding and centering the content properly

```
src > app > list-todos > list-todos.component.html > h1
1  <h1>My Todo's</h1>
2  <div class="container">
3      <table class="table">
4          <thead>
5              <tr>
6                  <th>Description</th>
7                  <th>TargetDate</th>
8                  <th>Is Completed</th>
9              </tr>
10         </thead>
11         <tbody>
12             <!--for (Todo todo : todos)
13         (for each element todo inside todos)-->
14             <tr *ngFor="let todo of todos">
15                 <td>{{todo.description}}</td>
16                 <!-- | means convert the previous object to this format-->
17                 <td>{{todo.targetDate | date | uppercase}}</td>
18                 <td>{{todo.done}}</td>
19             </tr>
20         </tbody>
21     </table>
22 </div>
```

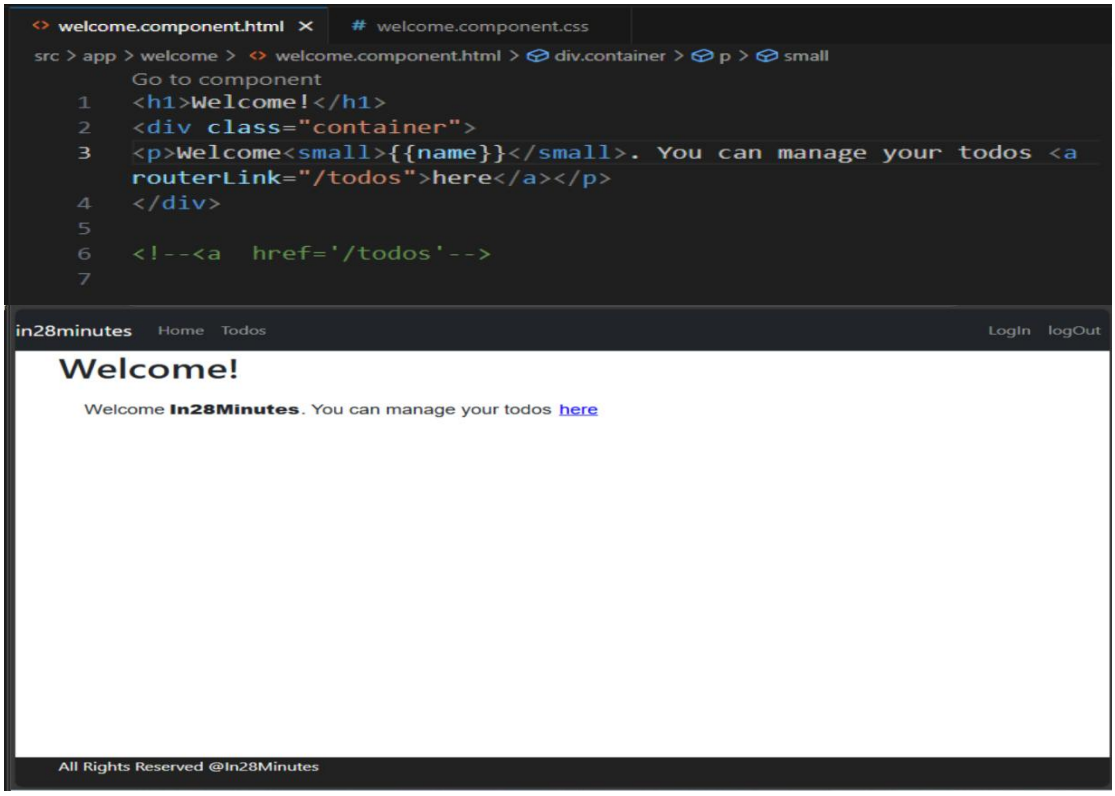


For the todo component, apply the same “container” on the table and include the class “table” for bootstrap gives a better centering on the content and give some style to the table.

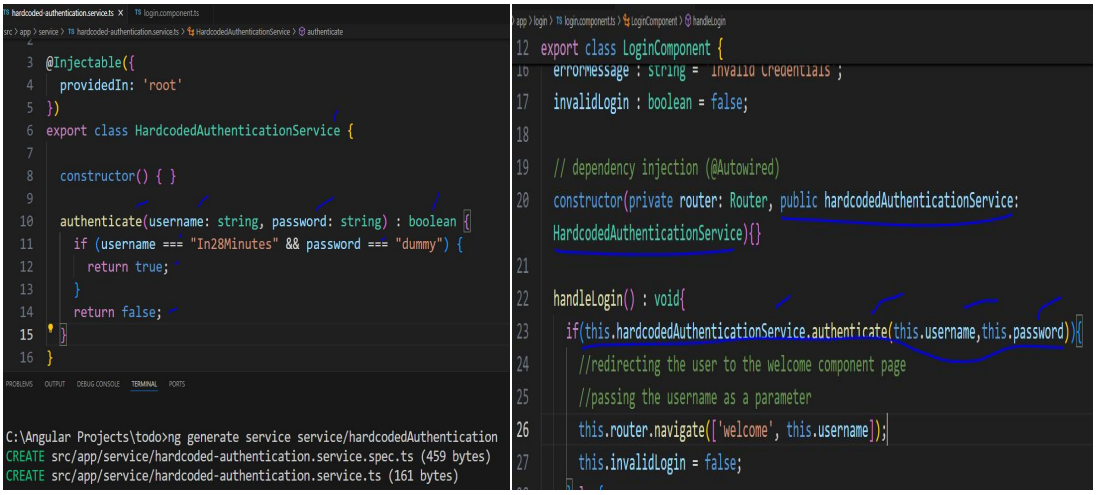
```
src > app > login > login.component.html > div.container > div.form > div > label
Go to component
1   <h1>Welcome to the Login page</h1>
2
3   <div class="container">
4       <div class="form">
5           <div class="alert alert-warning" *ngIf="invalidLogin">
            {{errorMessage}}</div>
6           <div>
7               <label for="username">Username:</label>
8               <input type="text" name="username" id="username"
                [(ngModel)]="username"> <!--value="{{username}}"-->
9               <label for="password">Password: </label>
10              <input type="password" name="password"
                id="password" [(ngModel)]="password">
11              <button (click)=handleLogin() class="btn
                btn-success">Login</button>
12          </div>
13      </div>
14  </div>
15  <!--adding meaning to the template, angular (*ngif) will show
    the content IF the invalidLogin is true, relating ngif to
    property -->
```



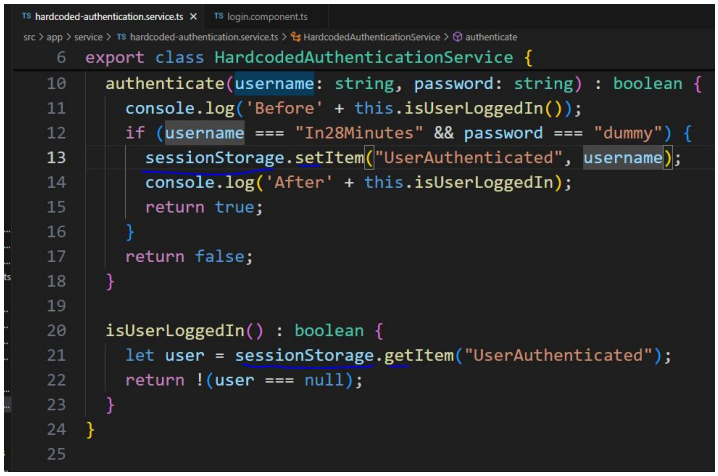
For the login component, apply the same container, the alert alert-warning for invalid login field and the btn btn-success for the button.

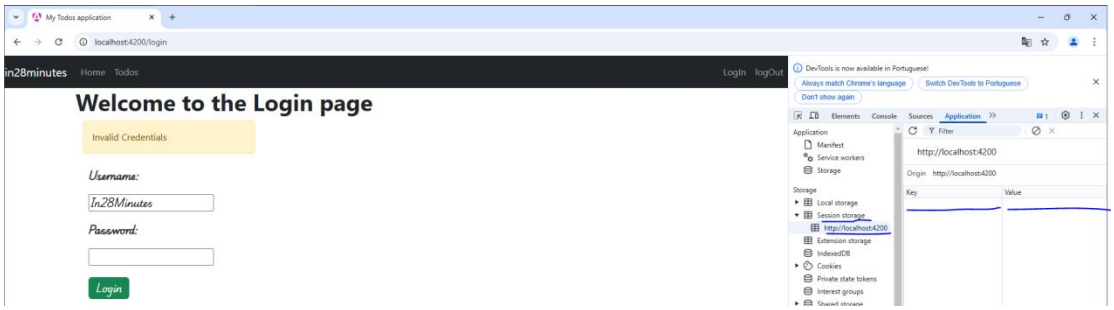


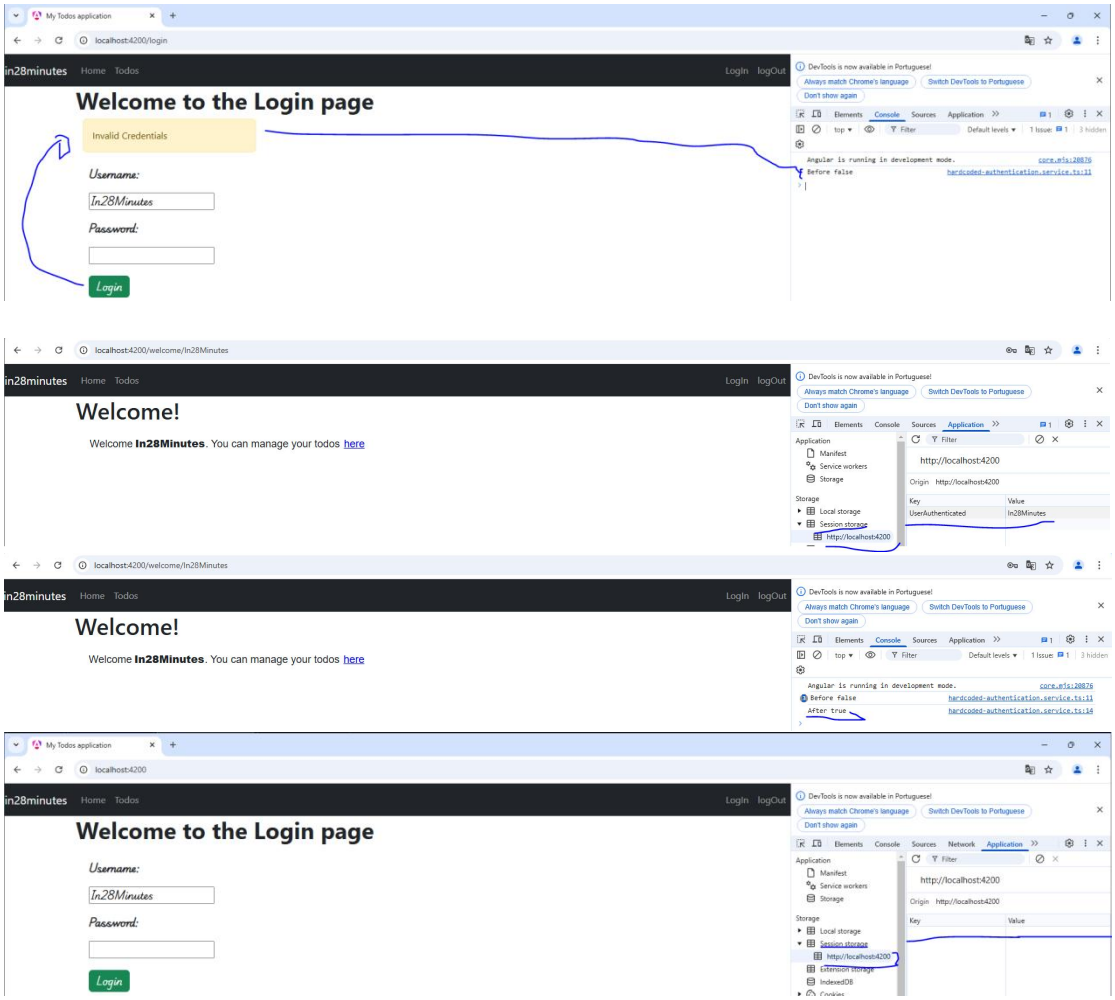
For the welcome component, apply the same container.



Now for improving the authentication logic, it is needed to create a service to center the authentication logic (business rules) **for all components**. For that, on the terminal, **ng generate service service/hardcodedAuthentication**. The service will contain the @Injectable decorator, to identify the class as a service, and be injectable/reusable in any part of the project through dependency injection on the component constructor. The logic used in the service is to verify if the username and password parameters are the expected ones, if it's okay, return true else false.





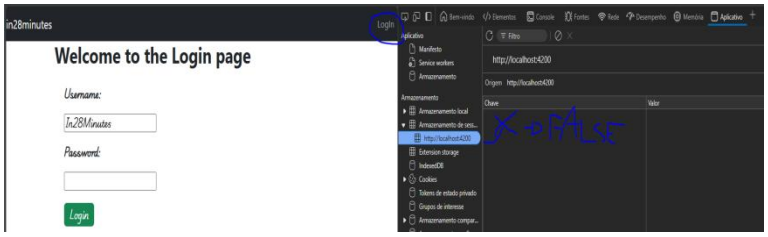


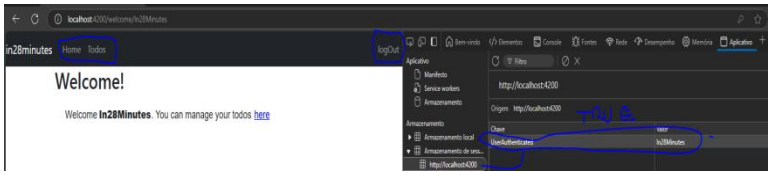
(after closing the browser)

A way to refine the authentication and make sure the user is logged in is to use the **session storage** feature from the browser. It allows us to store the username at browser storage while his session is open, after the browser is closed this data is erased, if we used the local storage, after the browser is closed that would still be there (less safe). The logic is that when the user clicks on the login button with an invalid login, the authentication service will check if the user is ‘saved’ on the session storage of the browser and return false on the console, now if the user enters with a valid login, the username is saved on the session storage and return true on the console. After closing the browser, the session is finished and the data is erased.

```typescript
import { Component } from '@angular/core';
import { RouterLink } from '@angular/router';
import { HardcodedAuthenticationService } from '../service/hardcoded-authentication.service';
import { NgIf } from '@angular/common';
@Component({
  selector: 'app-menu',
  imports: [RouterLink, NgIf],
  templateUrl: './menu.component.html',
  styleUrl: './menu.component.css'
})
export class MenuComponent {

  constructor(public hardcodedAuthenticationService: HardcodedAuthenticationService){}
}
```

```html
<ul class="navbar-nav">
    <li><a *ngIf="hardcodedAuthenticationService.isUserLoggedIn()" routerLink="/welcome/
    in28minutes" class="nav-link"> Home </a></li>
    <li><a *ngIf="hardcodedAuthenticationService.isUserLoggedIn()" routerLink="/todos"
    class="nav-link"> Todos </a></li>
</ul>

<ul class="navbar-nav navbar-collapse justify-content-end">
    <li><a *ngIf="!hardcodedAuthenticationService.isUserLoggedIn()" routerLink="/login"
    class="nav-link"> LogIn </a></li>
    <li><a *ngIf="hardcodedAuthenticationService.isUserLoggedIn()" routerLink="/logout"
    class="nav-link"> logOut </a></li>
</ul>
```





Once with the loggedIn state in hands, it makes sense, to render the menu links if the user is loggedIn or not. True: Home, Todos and Log out. Off: Login. For now, it's possible to reuse the loggedIn function (Service) directly on the template with directive ***ngIf** to render the links **according to the state**.

app.routes.ts

```
import { LogoutComponent } from './logout/logout.component';

//mapping the possible routes to render the correct component
export const routes: Routes = [
    {path: '', component: LoginComponent},
    {path: 'login', component: LoginComponent},
    {path: 'welcome/:name', component: WelcomeComponent},
    {path: 'todos', component: ListTodosComponent},
    {path: 'logout', component: LogoutComponent},
    {path: '**', component: ErrorComponent}
];
```

```
* History restored

Microsoft Windows [versão 10.0.22000.2538]
(c) Microsoft Corporation. Todos os direitos reservados.

C:\Angular Projects\todo>ng generate component logout --skip-import --standalone
CREATE src/app/logout/logout.component.html (22 bytes)
CREATE src/app/logout/logout.component.spec.ts (615 bytes)
CREATE src/app/logout/logout.component.ts (225 bytes)
CREATE src/app/logout/logout.component.css (0 bytes)
```

logout.component.html

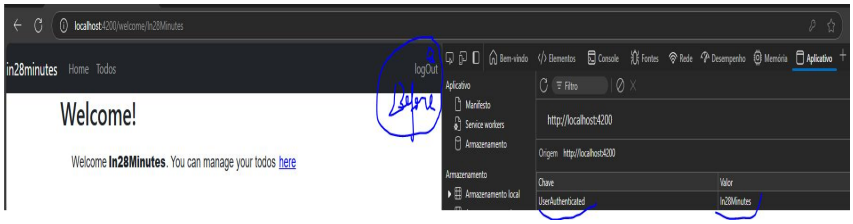
```
<h1>You are logged out</h1>
<div class="container">
        Thank You For Using Our Application.
</div>
```

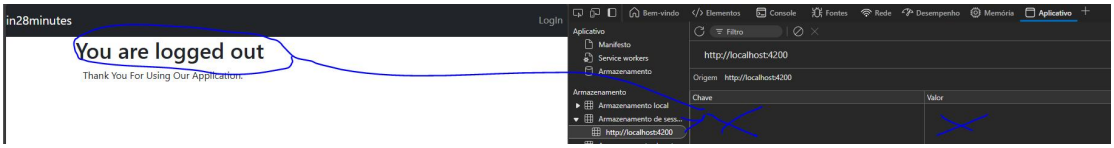
src/app/service/hardcoded-authentication.service.ts

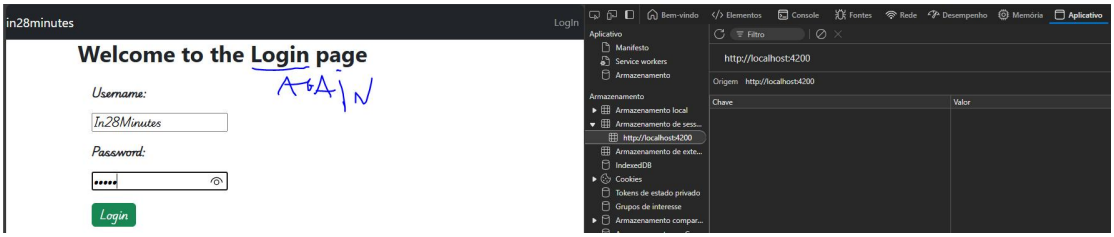
```
export class HardcodedAuthenticationService {

  logout(): void {
    sessionStorage.removeItem('UserAuthenticated');
  }
}
```

src/app/logout/logout.component.ts

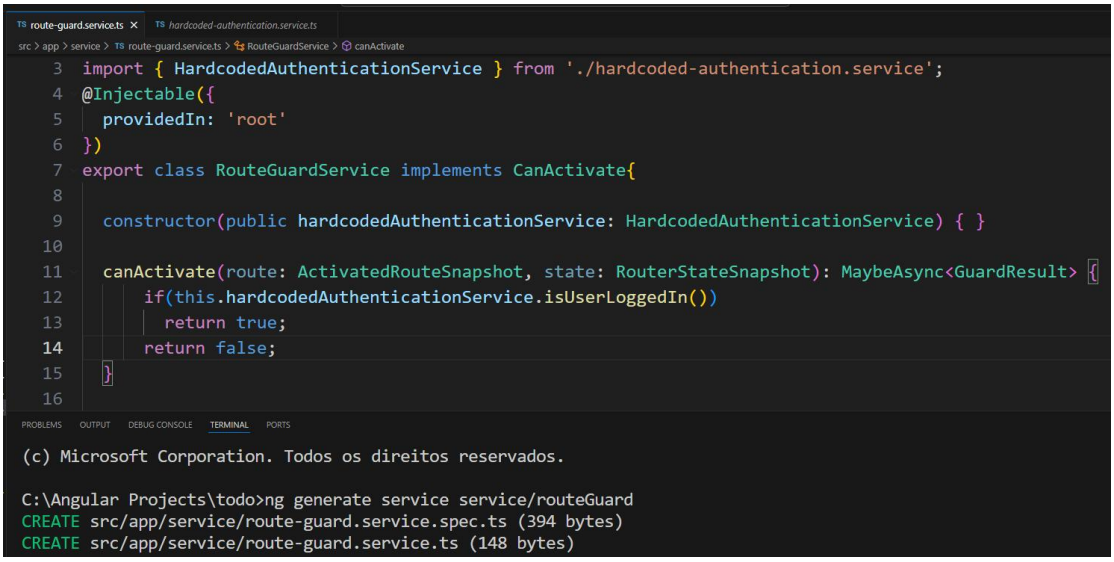
```
export class LogoutComponent implements OnInit {

  constructor(public hardcodedAuthenticationService: HardcodedAuthenticationService){}

  ngOnInit(){
    this.hardcodedAuthenticationService.logout();
  }

}
```







After completing the loggedIn, now it is important to deal with the log out, this control is being done by **sessionStorage** from the browser, if loggedIn means **data** on it, logged out means **no data** on it. This way, in the service one more function is created **to remove this data from the sessionStorage -> sessionStorage.removeItem('AuthenticatedUser')** and use it on the logout component.



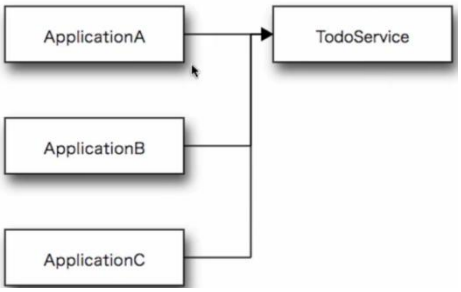
To finish the frontend part with Angular, it is needed to control the routes that the client can access based on the **condition** logged in || out, for that it will be created a specific service called route guard that will implement the

CanActivate interface and return true or false as result to be used in the **app.routes.ts** in the array -> {path:route, component: component, **canActivate:[RouteGuardService]**.

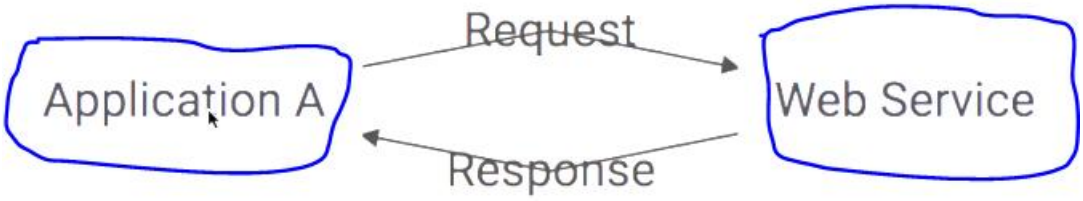
```
route-guard.service.ts  app.routes.ts
src > app > service > route-guard.service.ts > RouteGuardService > canActivate
})
export class RouteGuardService implements CanActivate{

  constructor(public hardcodedAuthenticationService: HardcodedAuthenticationService, public
  router:Router) { }

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): MaybeAsync<GuardResult> {
      if(this.hardcodedAuthenticationService.isUserLoggedIn())
        return true;
      this.router.navigate(['login']);
      return false;
  }

}
```

Improving it, if the user is not logged in and tries to access another routes, it is more important that he be redirected to the login page. For that, the **router reference** sorts it out-> **this.router.navigate([‘login’])**.

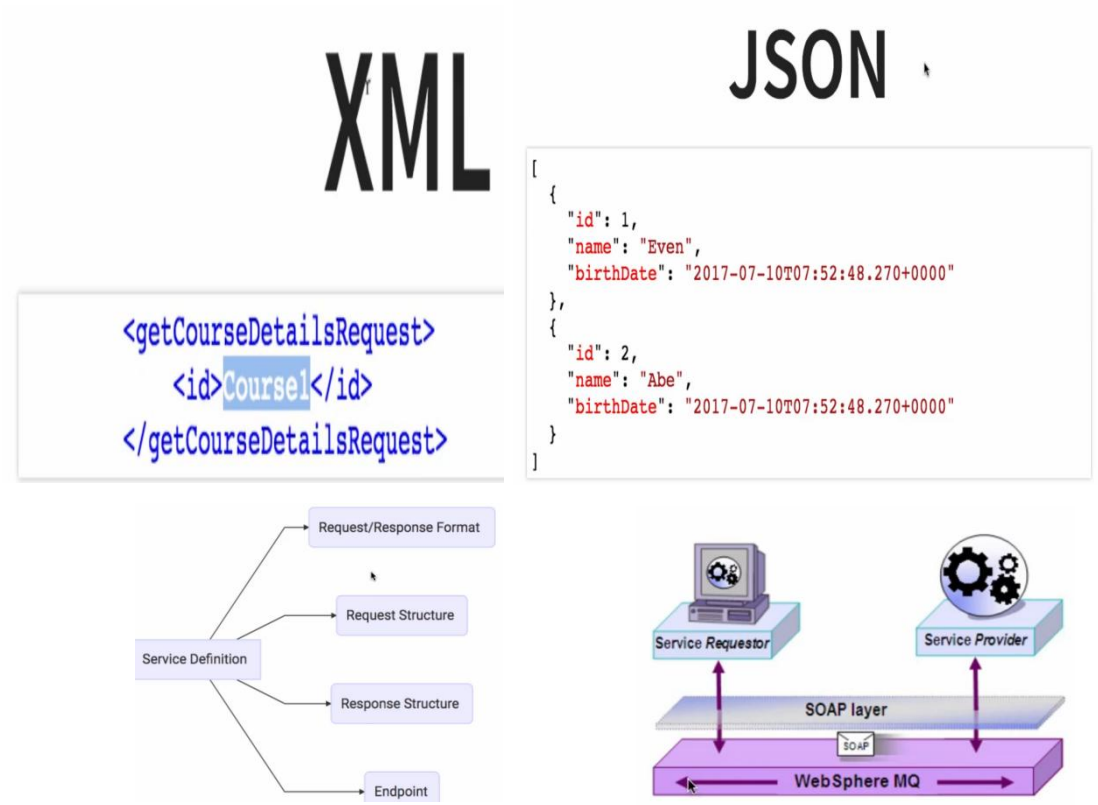
MicroServices



Web service is a software designed to support **interoperable machine-to-machine | app to app interaction over a network**, allowing that **different softwares (Java, CSharp, PHP) communicate** and exchange data without necessity of conversion of language | code. Working as a service provider or server, and the client as a service consumer.



The interactions between the apps requires an **input** to generate **processing** and delivering an **output** that can be summarized in **request and response**. For instance, the app A consumes from the web service a Ranga's todos list then the web service receives the **request**, look for it in the database and **respond** the customer with this list.

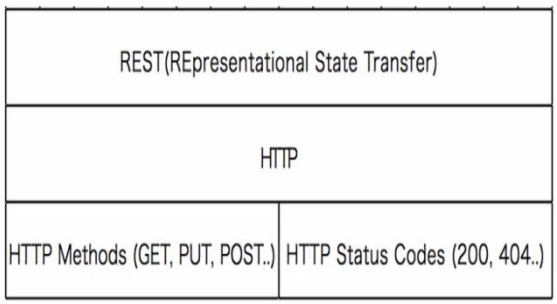


Even if the web service is developed in Java, it might be able to communicate with other platforms for that, two possible request & responses formats can be used:

XML (Extensible Markup Language) or JSON (JavaScript Object Notation).

Along with the format, the route / endpoint will be the way used by the customer to request / call the specific service. The transportation side uses the http protocol with a verb (get,post,put,delete) or by the co**m**munication **q**ueue.

MAKE BEST USE OF HTTP



KEY ABSTRACTION - RESOURCE

- A resource has an URI (Uniform Resource Identifier)
  - /user/Ranga/todos/1
  - /user/Ranga/todos
  - /user/Ranga
- A resource can have different representations
  - XML
  - HTML
  - JSON

EXAMPLE

- Create a User - POST /users
- Delete a User - DELETE /users/1
- Get all Users - GET /users
- Get one Users - GET /users/1

REST

- Data Exchange Format
  - No Restriction. JSON is popular
- Transport
  - Only HTTP
- Service Definition
  - No Standard. WADL/Swagger/...

REST -> Representational State Transfer

HTTP-> Hipertext Transfer Protocol

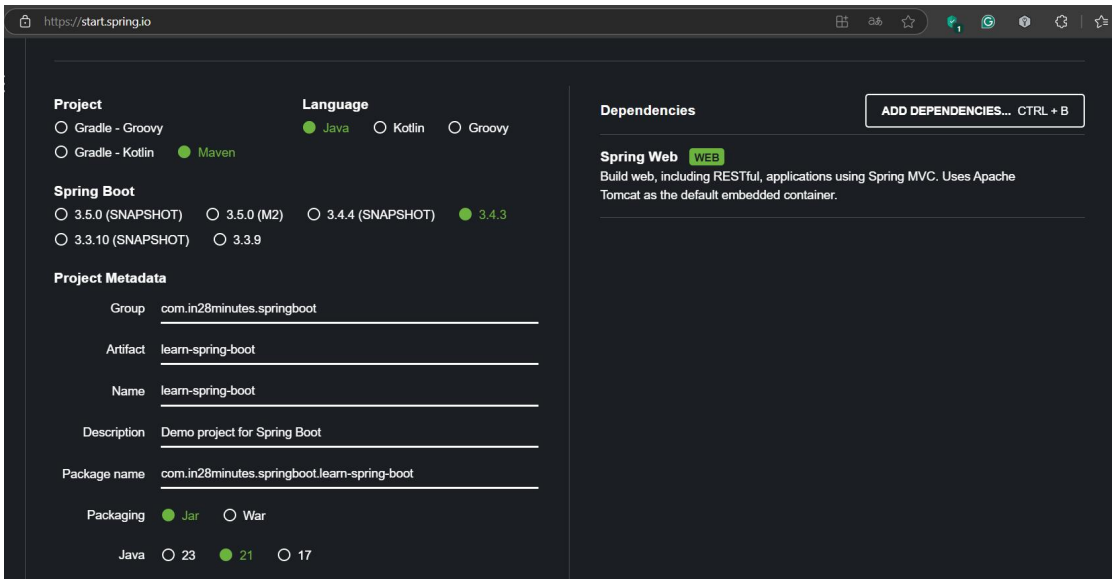
URL-> Uniform Resource Locator aka address/domain

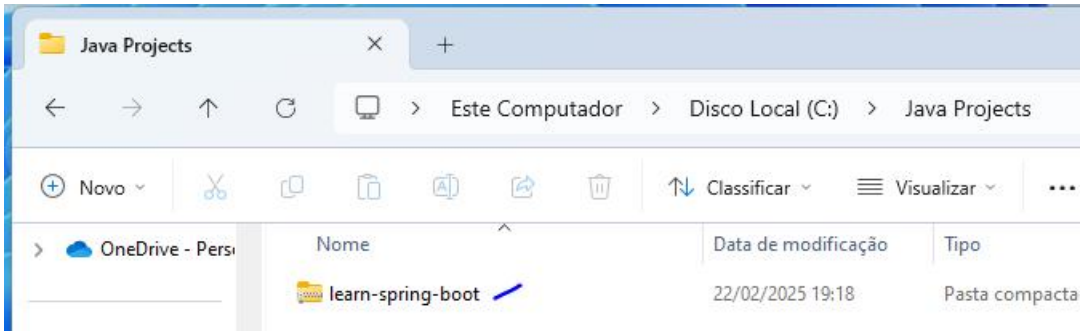
URI-> Uniform Resource Identifier

When navigating over the web, every time the consumer browser makes a request through an URL, it does a GET http request to receive the copy of the website (html) and the server responds sucessfully with a 200 status code. If this same user clicks over a button to register in the website, the browser will make a POST http request with some data in the body of the request like name, email, password and so on.

The REST idea is to reuse the http structure to share the content consumed by the client and the operations with it, using URIs as routes along with the URL to exposure the service. For instance:
/user/Ranga/todos/1-> bring me the first todo of the Ranga's list todo.
/user/Ranga/todos/-> bring me the Ranga's list todo.
/user/Ranga-> bring the user data or the content that this user has access.
Before going ahead with the section 5 it is advised to make an overall review about Spring and JPA.

Apprendix:
Why, What are the goals, how it works, comparison with Spring and Spring MVC.
Highlights: Spring intialzr, starter projects, auto configuration, developer tools, actuator.
Before Spring **Boot** in 2016, every new project in spring, it would be needed to configure:
1. Dependency Management (pom.xml)
2. Define Web App Configuration (web.xml)
3. Manage Spring Beans (context.xml)
4. Deploy of non-functional requirements (nfrs).
Creating a project with Spring intialzr:
start.spring.io





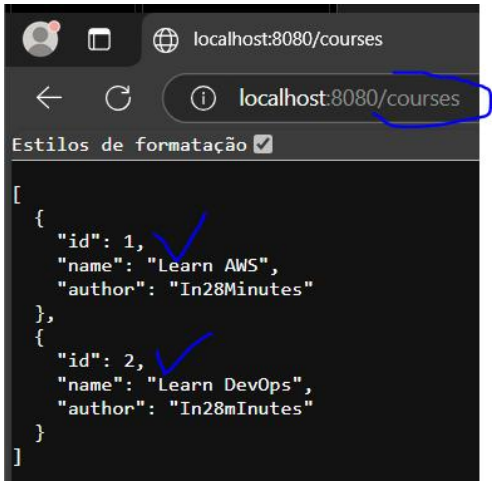
The idea is to make a rest api to bring back a list of courses like this [{id:1,name:'learn AWS', author:'In28Minutes'}]. After importing the project, create a controller class like courseController and put @RestController to identify the class as a Rest API controller in fact, then create a method to return the list of courses and include the uri/endpoint "/courses" using the @RequestMapping("/courses"). Being needed to have the class Course to represent this object in real life.

CourseController.java

```java
package com.in28minutes.springboot.learn_spring_boot;

import java.util.Arrays;
import java.util.List;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class CourseController {

    @RequestMapping("/courses")
    public List<Course> retrieveAllCourses(){
        return Arrays.asList(
                new Course(1,"Learn AWS", "In28Minutes"),
                new Course(2,"Learn DevOps", "In28mInutes")
                );
    }
}
```

*Course.java

```java
public class Course {
    private long id;
    private String name;
    private String author;
    public Course(long id, String name, String author) {
        this.id = id;
        this.name = name;
        this.author = author;
    }
    public long getId(){
        return id;
    }
    public String getName() {
        return name;
    }
    public String getAuthor() {
        return author;
    }
    @Override
    public String toString() {
        return "Course [id=" + id + ", name=" + name + ", author=" + author + "]";
    }
```



The goal of the spring boot is **quickly** building **production-ready** apps:

Quickly:

- Spring initialzr-> fast project creation
- Spring Boot starter projects -> fast dependency grouping
- Spring Boot Auto Configuration-> Automatic configurations like tomcat, h2 database, @beans (injection dependency) and others based on the dependencies.
- Spring Boot DevTools -> Running managing of the server like restart the server automatically if there is code change to increase the productivity. Except by insertion of new dependencies on pom.xml.

Production-ready features:

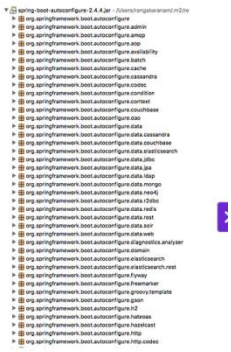
- Logging(reports on terminal)
- Profiles through properties file for each environment (dev,qa,stage,prod).
- Monitoring (Spring Boot Actuator)-> hardware and software managing on the cloud like metrics of more/less memory, hard disk space, using of cpu;

Exploring Spring Boot Starter Projects

- I need a lot of frameworks to build application features:
  - **Build a REST API**: I need Spring, Spring MVC, Tomcat, JSON conversion...
  - **Write Unit Tests**: I need Spring Test, JUnit, Mockito, ...
- How can I group them and make it easy to build applications?
  - **Starters**: Convenient **dependency descriptors** for diff. features
- **Spring Boot** provides variety of starter projects:
  - **Web Application & REST API** - Spring Boot Starter Web (spring-webmvc, spring-web, spring-boot-starter-tomcat, spring-boot-starter-json)
  - **Unit Tests** - Spring Boot Starter Test
  - **Talk to database using JPA** - Spring Boot Starter Data JPA
  - **Talk to database using JDBC** - Spring Boot Starter JDBC
  - **Secure your web application or REST API** - Spring Boot Starter Security

Exploring Spring Boot Auto Configuration

- I need **lot of configuration** to build Spring app:
  - Component Scan, DispatcherServlet, Data Sources, JSON Conversion, ...
- How can I simplify this?
  - **Auto Configuration: Automated configuration** for your app
    - **Decided** based on:
      - Which frameworks are in the Class Path?
      - What is the existing configuration (Annotations etc)?
- **Example**: Spring Boot Starter Web
  - Dispatcher Servlet (`DispatcherServletAutoConfiguration`)
  - Embedded Servlet Container - Tomcat is the default (`EmbeddedWebServerFactoryCustomizerAutoConfiguration`)
  - Default Error Pages (`ErrorMvcAutoConfiguration`)
  - Bean<->JSON (`JacksonHttpMessageConvertersConfiguration`)



161. Step 07 - Understanding Spring Boot Magic - Auto Configuration

localhost:8080/courses/10

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

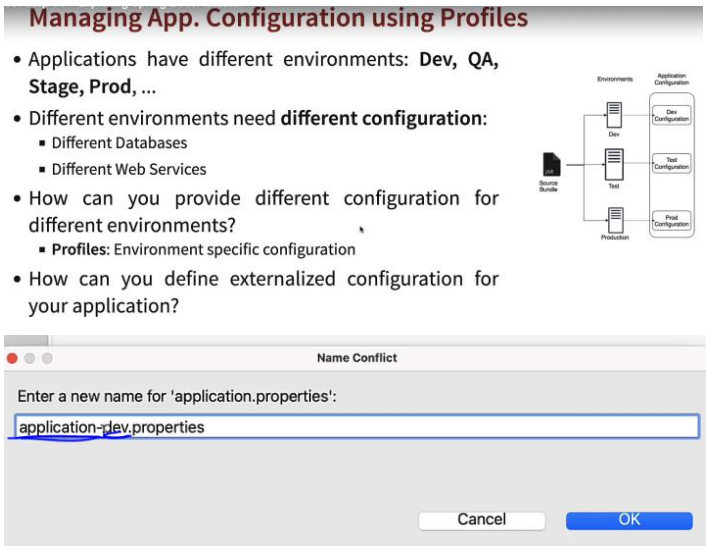
Tue Jul 05 11:18:44 IST 2022
There was an unexpected error (type=Not Found, status=404).

```
1 logging.level.org.springframework=debug
```

Starter projects will group several dependencies at once and based on it, the spring boot auto configuration will configure these dependencies to work without manual configuration by the developer like an error page when the endpoint is not mapped by the controller, but it can also be overridden in the

application.properties like debug (show the operational processes) the spring initialization until the server is up and ready.

```
<dependencies>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
    </dependency>
```




```
1 logging.level.org.springframework=trace
```


```
1 logging.level.org.springframework=info
```

DEFAULT

```
*application.properties × | notes.txt | application-dev.properties | application-prod.properties
1 logging.level.org.springframework=debug
2 spring.profiles.active=prod
```

```
=========|_|==============|___/=/_/_/_/
:: Spring Boot ::                (v3.0.0-M3)

2022-07-05T11:42:54.139+05:30  INFO 37096 --- [  restartedMa
2022-07-05T11:42:54.139+05:30  INFO 37096 --- [  restartedMa

2022-07-05T11:43:32.139+05:30 TRACE 37096 --- [  restartedMain] o.s.b.f.s.DefaultListab
2022-07-05T11:43:32.139+05:30  INFO 37096 --- [  restartedMain] c.i.s.l.LearnSpringBoot
```

Levels:

Trace

Debug

Info

Warning

Error

Off

To specify the environment, create a new file with the environment name application-environmentname.properties. So for example create two files application-dev.properties and application-prod.properties, on dev type logging.level.org.springframework=trace will bring trace info and below, on prod type logging.level.org.springframework= info will bring info and below.

Supposing that we are consuming an external webservice called currency-service, then it is needed to create a few properties to **configure** this connection according to the environment.

Application.properties

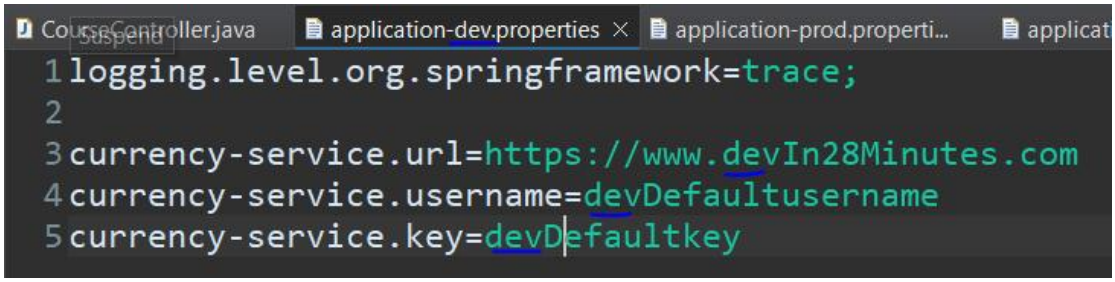
```
spring.application.name=learn-spring-boot
spring.profiles.active=dev

currency-service.url=https://www.In28Minutes.com
currency-service.username=defaultusername
currency-service.key=defaultkey
```

On the application.properties type:

currency-service.key=defaultkey

* prefix.propertyName=value

On the application-dev.properties type:

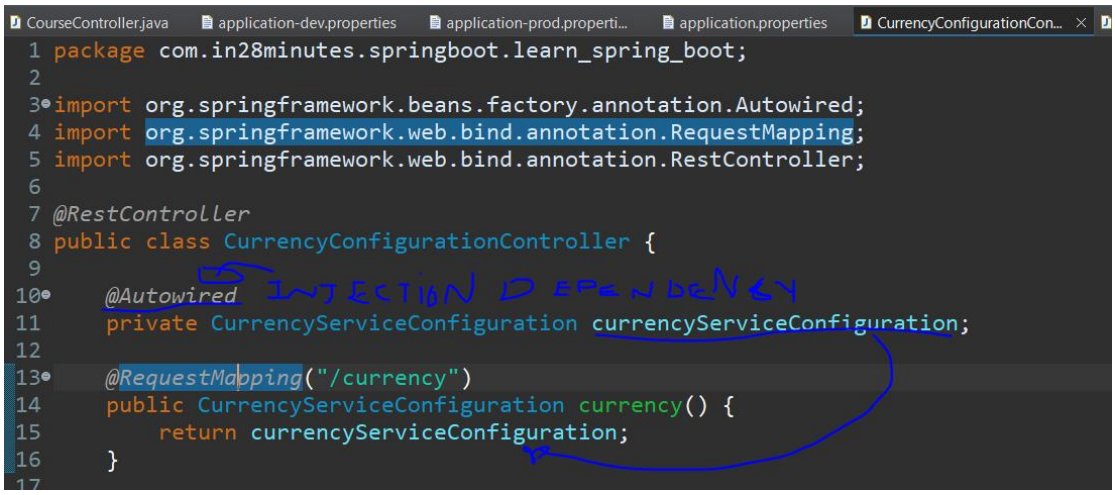


```java
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;

@ConfigurationProperties(prefix="currency-service")
@Component
public class CurrencyServiceConfiguration {

    private String url;
    private String username;
    private String key;

    public String getUrl() {
        return url;
    }

    public void setUrl(String url) {
        this.url = url;
    }
```

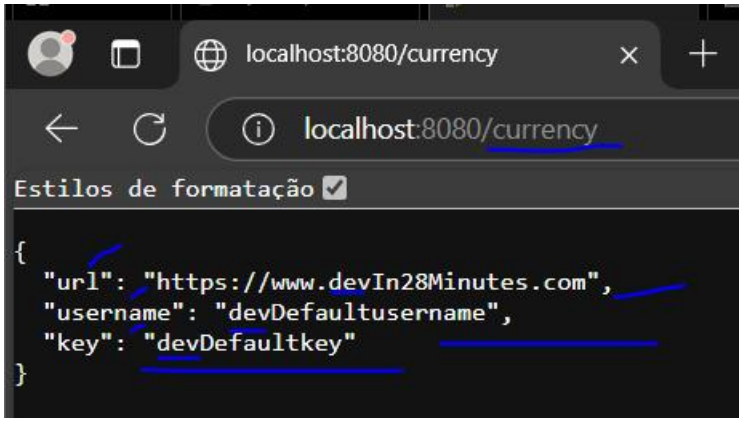
Create a new class called CurrencyServiceConfiguration with these properties along with their getters and setters. As Spring features, the annotations:

@ConfigurationProperties(prefix="currency-service") -> This annotation, identifies the class as the one with the properties of the webservice with prefix currency-service

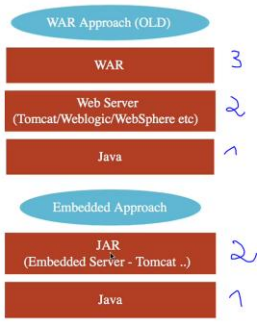
@Component -> This annotation, identifies the class as a Spring Component, making possible to make a bean dependency injection (@Autowired).



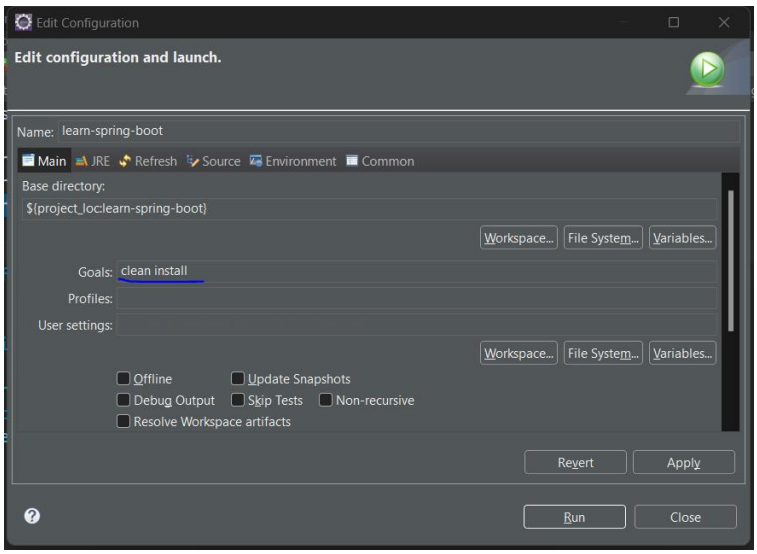
To test this approach, it is needed to create a controller to expose it. Using the dependency injection of the configuration class to return this properties filled in.



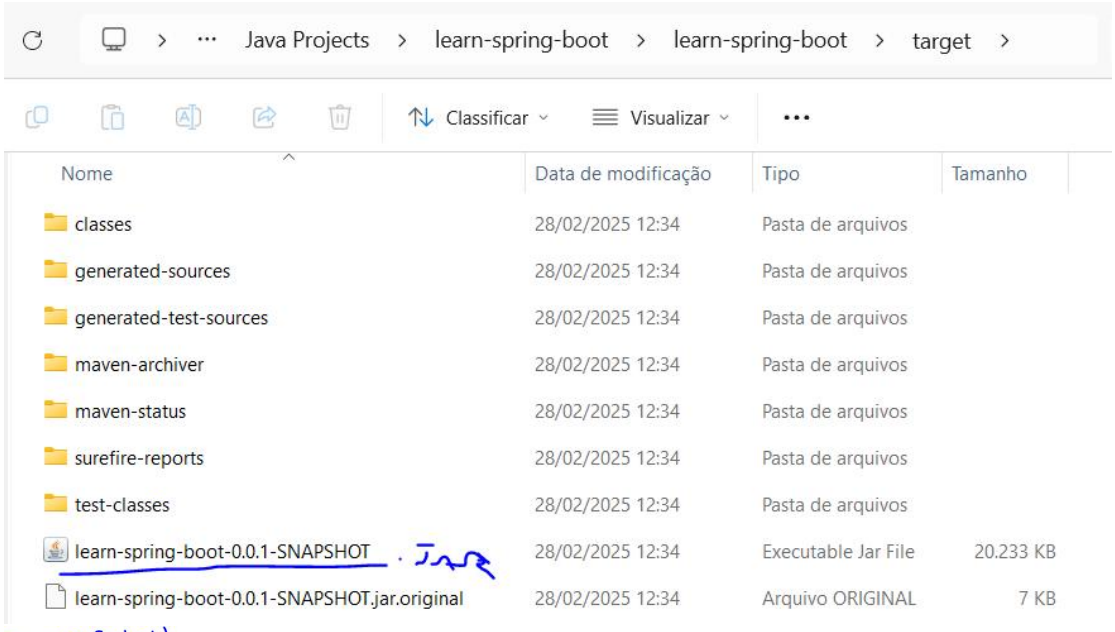
Embedded servers



*War-> Web ARchive (zipped project)



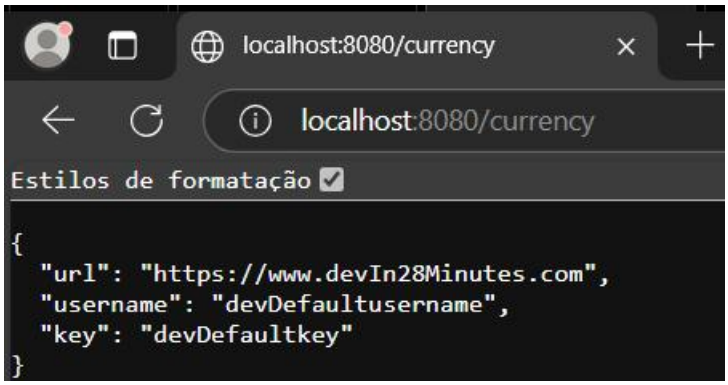

```
[INFO] Installing C:\Java Projects\learn-spring-boot\learn-spring-boot\target\learn-spring-boot-0.0.1-SNAPSHOT.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  22.553 s
[INFO] Finished at: 2025-02-28T12:34:57-03:00
[INFO] ------------------------------------------------------------------------
```



POWERSHELL

```
PS C:\Java Projects\learn-spring-boot\learn-spring-boot\target> java -jar learn-spring-boot-0.0.1-SNAPSHOT.jar
```

```
2025-02-28T12:49:55.724-03:00 TRACE 4740 --- [learn-spring-boot] [           main] o.s.b.f.s.DefaultListableBeanFactory     : Returning cached instance of s
ingleton bean 'welcomePageHandlerMapping'
2025-02-28T12:49:55.724-03:00 TRACE 4740 --- [learn-spring-boot] [           main] o.s.b.f.s.DefaultListableBeanFactory     : Returning cached instance of s
ingleton bean 'welcomePageNotAcceptableHandlerMapping'
2025-02-28T12:49:55.724-03:00 TRACE 4740 --- [learn-spring-boot] [           main] o.s.b.f.s.DefaultListableBeanFactory     : Returning cached instance of s
ingleton bean 'requestMappingHandlerMapping'
2025-02-28T12:49:55.724-03:00 TRACE 4740 --- [learn-spring-boot] [           main] o.s.b.f.s.DefaultListableBeanFactory     : Returning cached instance of s
ingleton bean 'viewControllerHandlerMapping'
2025-02-28T12:49:55.724-03:00 TRACE 4740 --- [learn-spring-boot] [           main] o.s.b.f.s.DefaultListableBeanFactory     : Returning cached instance of s
ingleton bean 'beanNameHandlerMapping'
2025-02-28T12:49:55.724-03:00 TRACE 4740 --- [learn-spring-boot] [           main] o.s.b.f.s.DefaultListableBeanFactory     : Returning cached instance of s
ingleton bean 'routerFunctionMapping'
2025-02-28T12:49:55.724-03:00 TRACE 4740 --- [learn-spring-boot] [           main] o.s.b.f.s.DefaultListableBeanFactory     : Returning cached instance of s
ingleton bean 'resourceHandlerMapping'
2025-02-28T12:49:55.724-03:00 TRACE 4740 --- [learn-spring-boot] [           main] o.s.b.f.s.DefaultListableBeanFactory     : Returning cached instance of s
ingleton bean 'defaultServletHandlerMapping'
2025-02-28T12:49:55.740-03:00  INFO 4740 --- [learn-spring-boot] [           main] c.i.s.l.LearnSpringBootApplication       : Started LearnSpringBootApplica
tion in 4.747 seconds (process running for 5.565)
2025-02-28T12:49:55.740-03:00 TRACE 4740 --- [learn-spring-boot] [           main] o.s.b.f.s.DefaultListableBeanFactory     : Returning cached instance of s
ingleton bean 'applicationAvailability'
2025-02-28T12:49:55.740-03:00 DEBUG 4740 --- [learn-spring-boot] [           main] o.s.b.a.ApplicationAvailabilityBean      : Application availability state
 LivenessState changed to CORRECT
2025-02-28T12:49:55.740-03:00 TRACE 4740 --- [learn-spring-boot] [           main] o.s.b.f.s.DefaultListableBeanFactory     : Returning cached instance of s
ingleton bean 'applicationAvailability'
2025-02-28T12:49:55.740-03:00 DEBUG 4740 --- [learn-spring-boot] [           main] o.s.b.a.ApplicationAvailabilityBean      : Application availability state
 ReadinessState changed to ACCEPTING_TRAFFIC
```



- How do you deploy your application?
  - Step 1 : Install Java
  - Step 2 : Install Web/Application Server
    - Tomcat/WebSphere/WebLogic etc
  - Step 3 : Deploy the application WAR (Web ARchive)
    - This is the OLD **WAR** Approach
    - Complex to setup!
- **Embedded Server** - Simpler alternative
  - Step 1 : Install Java
  - Step 2 : Run **JAR** file

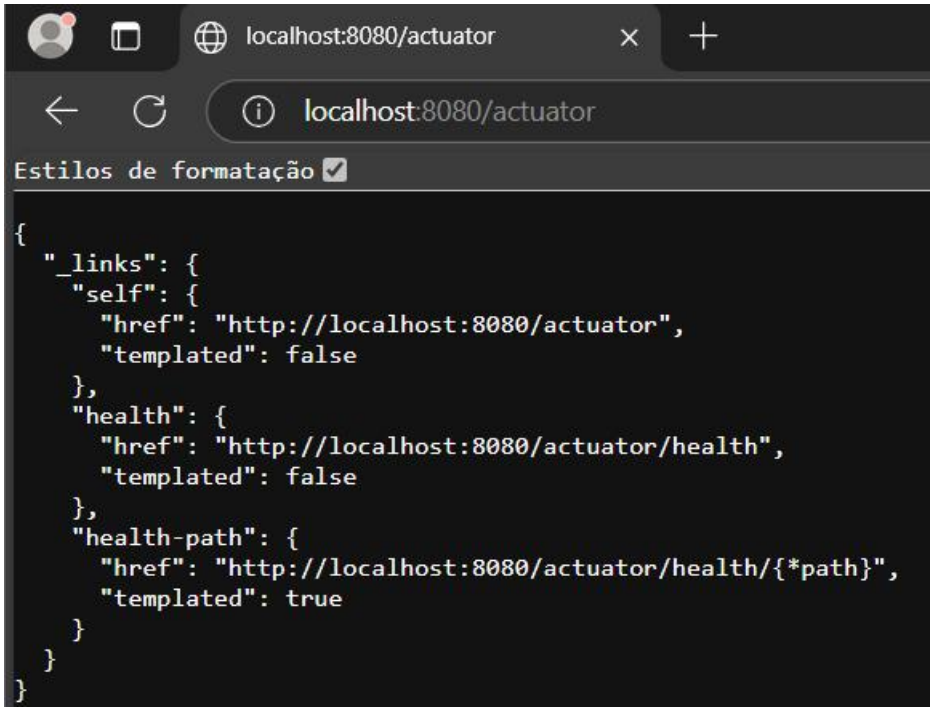
When using this command **clean install** the system will erase any previous outdated version, then build, run and test a new one with the new code, and the end it generates an executable of this new version as .jar (.apk,.exe). Once, the project is exported, it can be executed on the terminal. As alternative embedded servers:spring-boot-starter-**tomcat/jetty/undertow**

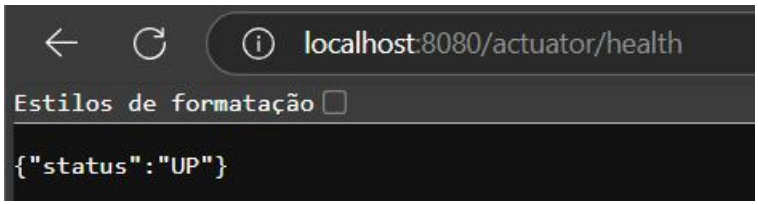
Spring boot Actuator will help to monitor and manage the application on the cloud and it will expose a few endpoints:

- Beans -> List of Spring beans
- Health-> Information of running status like up and running or up and error
- Metrics-> Using of memory, space, cpu, network, demand
- Mapping -> Details around request mapping

Pom.xml:

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

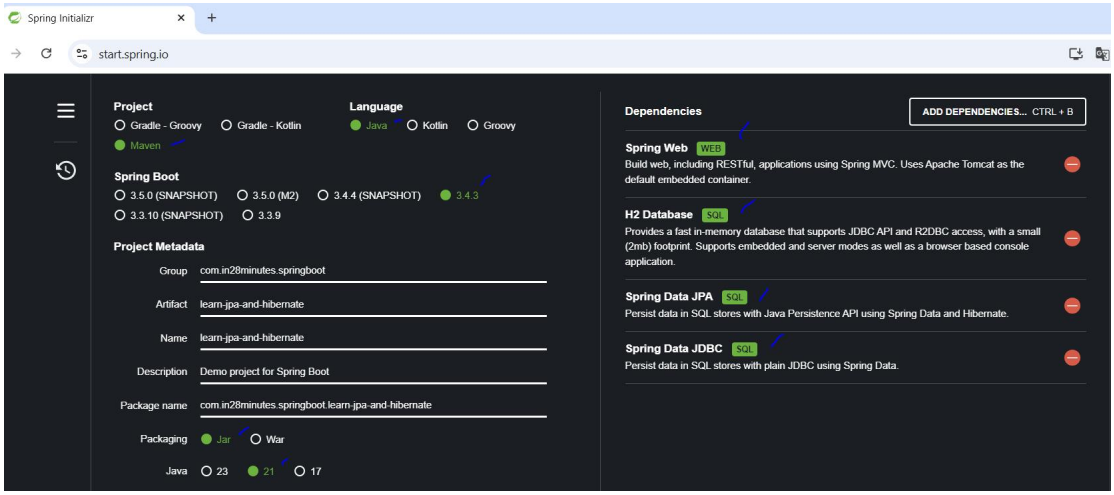




To edit which endpoints that are available or see them all (*), on the application.properties include the line: management.endpoints.web.exposure.include= health,metrics or management.endpoints.web.exposure.include=*.

Understanding Spring Boot vs Spring MVC vs Spring

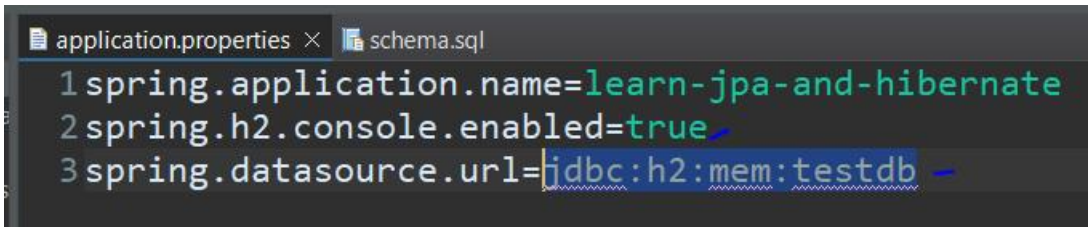
- **Spring Boot vs Spring MVC vs Spring**: What's in it?
  - **Spring Framework**: Dependency Injection
    - @Component, @Autowired, Component Scan etc..
    - Just Dependency Injection is NOT sufficient (You need other frameworks to build apps)
      - **Spring Modules and Spring Projects**: Extend Spring Eco System
        - Provide good integration with other frameworks (Hibernate/JPA, JUnit & Mockito for Unit Testing)
  - **Spring MVC** (Spring Module): Simplify building web apps and REST API
    - Building web applications with Struts was very complex
    - @Controller, @RestController, @RequestMapping("/courses")
  - **Spring Boot** (Spring Project): Build **PRODUCTION-READY** apps **QUICKLY**
    - **Starter Projects** - Make it easy to build variety of applications
    - **Auto configuration** - Eliminate configuration to setup Spring, Spring MVC and other frameworks!
    - Enable non functional requirements (NFRs):
      - **Actuator**: Enables Advanced Monitoring of applications
      - **Embedded Server**: No need for separate application servers!
      - Logging and Error Handling
      - Profiles and ConfigurationProperties




```
Bootstrapping Spring Data JDBC repositories in DEFAULT mode.
Finished Spring Data repository scanning in 16 ms. Found 0 JDBC repository interfaces.
Multiple Spring Data modules found, entering strict repository configuration mode
Bootstrapping Spring Data JPA repositories in DEFAULT mode.
Finished Spring Data repository scanning in 3 ms. Found 0 JPA repository interfaces.
Tomcat initialized with port 8080 (http)
Starting service [Tomcat]
Starting Servlet engine: [Apache Tomcat/10.1.36]
Initializing Spring embedded WebApplicationContext
Root WebApplicationContext: initialization completed in 3286 ms
HikariPool-1 - Starting...
HikariPool-1 - Added connection conn0: url=jdbc:h2:mem:33910671-dfcb-42fc-88f0-5e9fa979b079
HikariPool-1 - Start completed.
HHH000204: Processing PersistenceUnitInfo [name: default]
HHH000412: Hibernate ORM core version 6.6.8.Final
HHH000026: Second-level cache disabled
```

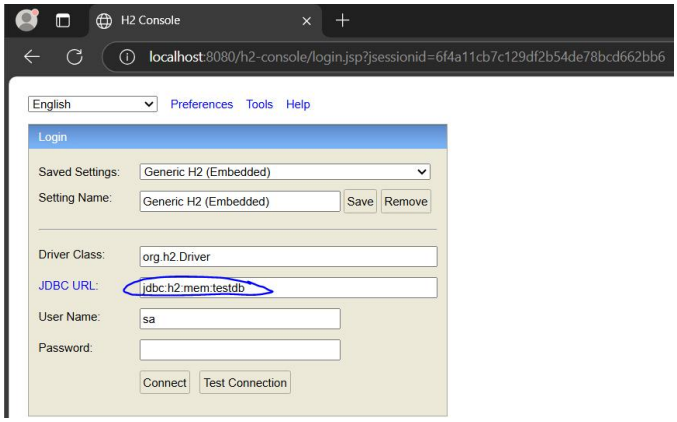
From this moment ahead, it's time to have an appendix section for manipulating specifically the database using **JPA, Hibernate and JDBC**. Creating a project with the dependencies: spring web, h2 database, spring

data JPA, spring data JDBC. With these dependencies loaded, the creation and configuration of the database is auto configured, the tomcat server, the jpa and jdbc modules too.

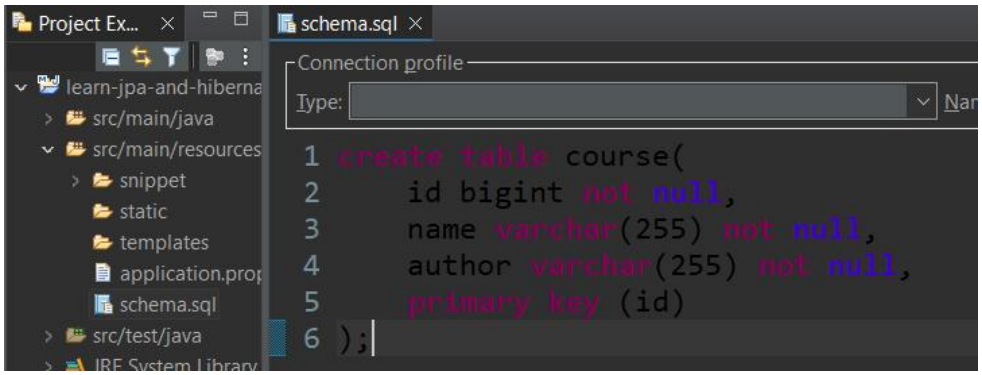


To access the database through console and specify the database url, it is needed to include two properties on application.properties:

- Spring.h2.console.enabled=true (Access database through browser)
- Spring.datasource.url=jdbc:h2:mem:testdb (specify the database url)



Once configured, it can be accessed by the browser through the url localhost:8080/h2-console.



Also would be interesting if this database already had a table like course.For that, it is needed to create a file called schema.sql containing the following code:

Create table course (

Id bigint not null, // bigint is the long in java

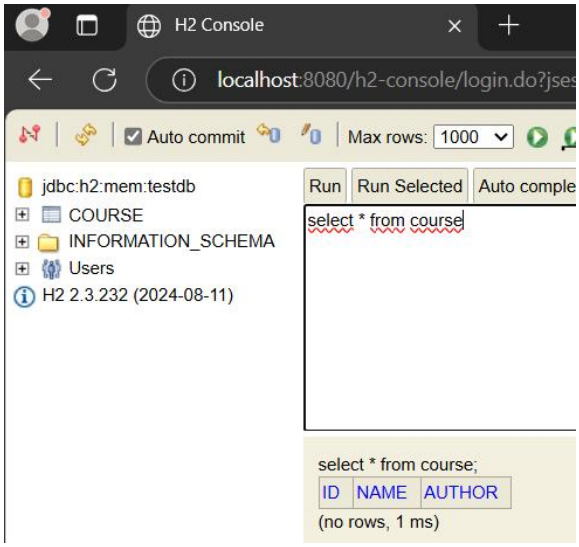
Name varchar(255) not null, //varchar is the string in java

Author varchar(255) not null,

Primary key(id) //defining the primary key of the table

);

Once the server is restarted, this is the result:



Jdbc (raw, without spring **boot)**

```
Public void deleteTodo(int id){
        PreparedStatement st = null;
        Try{
          St = db.conn.prepareStatement(“delete from todo where id=?”);
          St.setInt(1, id);
          St.execute();
        }catch(SQLException e){
                Logger.fatal(“Query failed : ”+ e);
        } finally {
                If ( st != null){
                Try {St.close();}
                Catch(SqlException e){}
                }
        }
}
```

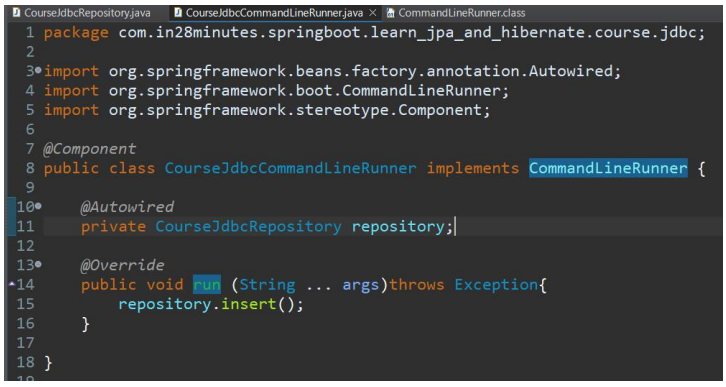
Spring **data** jdbc

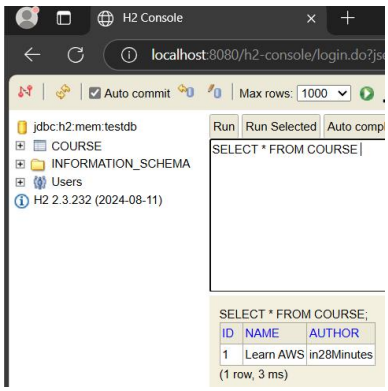
Public void deleteTodo(int id){

JdbcTemplate.update(“delete from todo where id = ?”, id);

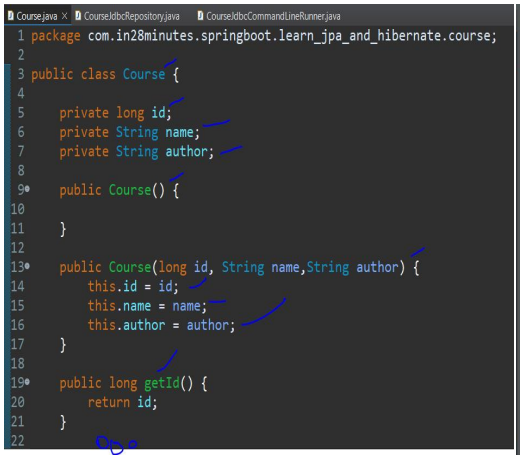
}

Spring Jdbc x Jdbc, both can be used for manipulating data on database with a lot of sql code, but with spring, **less java code is written**. Based on it, it is needed to create a class containing the object **jdbcTemplate**, the sql statement and a method to execute the operation, then this class is considered a repository (data access layer).

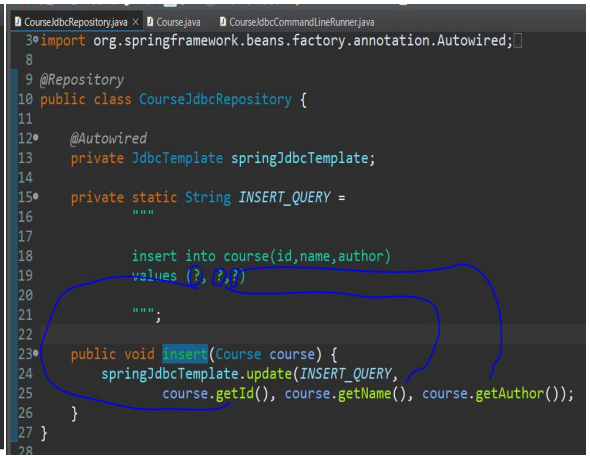




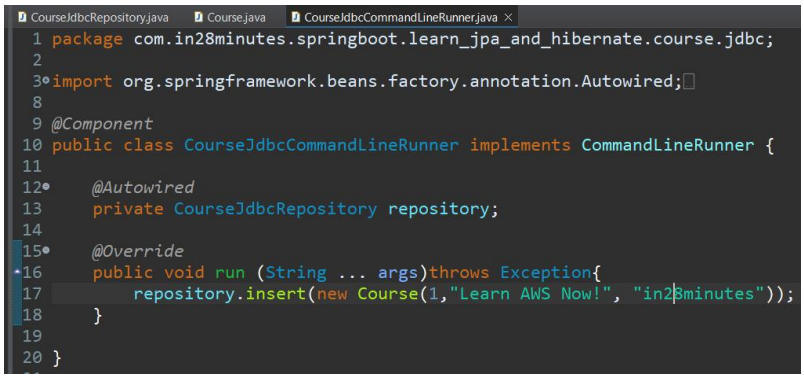
For executing it when the application starts, the interface commandLineRunner needs to be implemented by a new class as a spring component and call the operation from the repository. This way, the app starts creating the database in memory, with the table course created and **with a register inserted**.



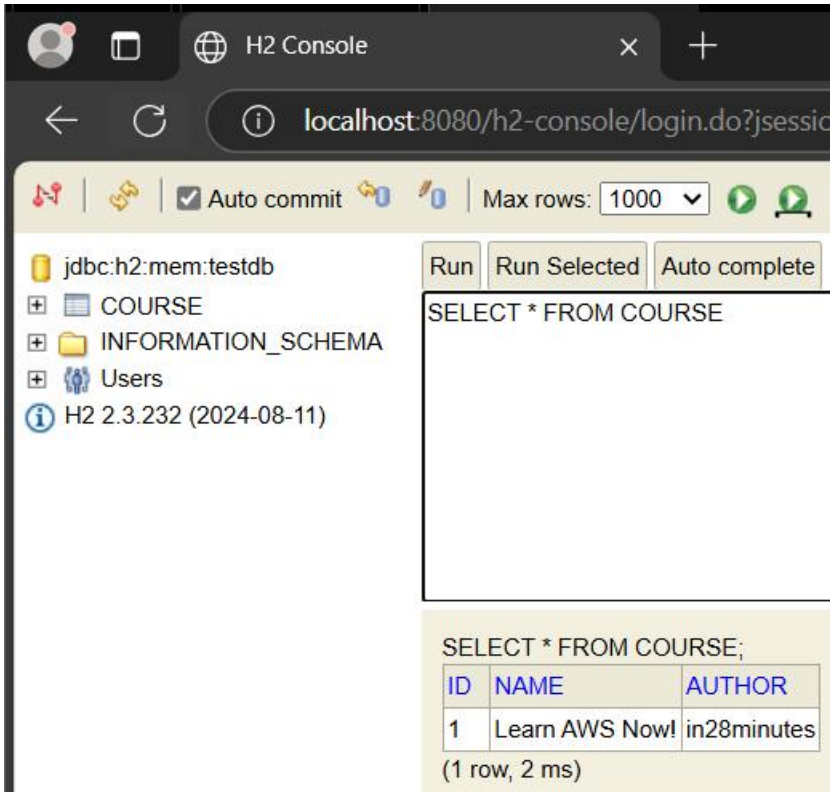
```java
package com.in28minutes.springboot.learn_jpa_and_hibernate.course;

public class Course {

    private long id;
    private String name;
    private String author;

    public Course() {

    }

    public Course(long id, String name,String author) {
        this.id = id;
        this.name = name;
        this.author = author;
    }

    public long getId() {
        return id;
    }
```



```java
import org.springframework.beans.factory.annotation.Autowired;

@Repository
public class CourseJdbcRepository {

    @Autowired
    private JdbcTemplate springJdbcTemplate;

    private static String INSERT_QUERY =
            """

            insert into course(id,name,author)
            values (?, ?,?)

            """;

    public void insert(Course course) {
        springJdbcTemplate.update(INSERT_QUERY,
                course.getId(), course.getName(), course.getAuthor());
    }
}
```



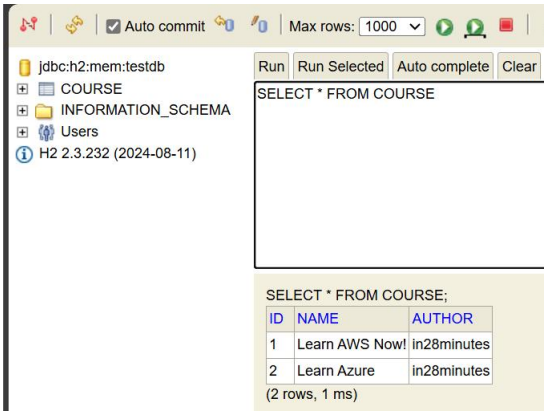
```java
package com.in28minutes.springboot.learn_jpa_and_hibernate.course.jdbc;

import org.springframework.beans.factory.annotation.Autowired;

@Component
public class CourseJdbcCommandLineRunner implements CommandLineRunner {

    @Autowired
    private CourseJdbcRepository repository;

    @Override
    public void run (String ... args)throws Exception{
        repository.insert(new Course(1,"Learn AWS Now!", "in28minutes"));
    }

}
```



Replacing the hardcoded values on the insert statement, it is needed to create a class that will represent this course containing the attributes. On the repository class replace the values by question mark symbols, making the insert method receive the new object course, inputing the get methods separated by coma in update method to represent these questions marks positioning. Finally, update the run method on commandLineRunner, passing a course instance.

```
    private static String DELETE_QUERY =
            """

            delete from course where id = ?

            """;

    public void deleteById(long id) {
        springJdbcTemplate.update(DELETE_QUERY, id);
    }
```

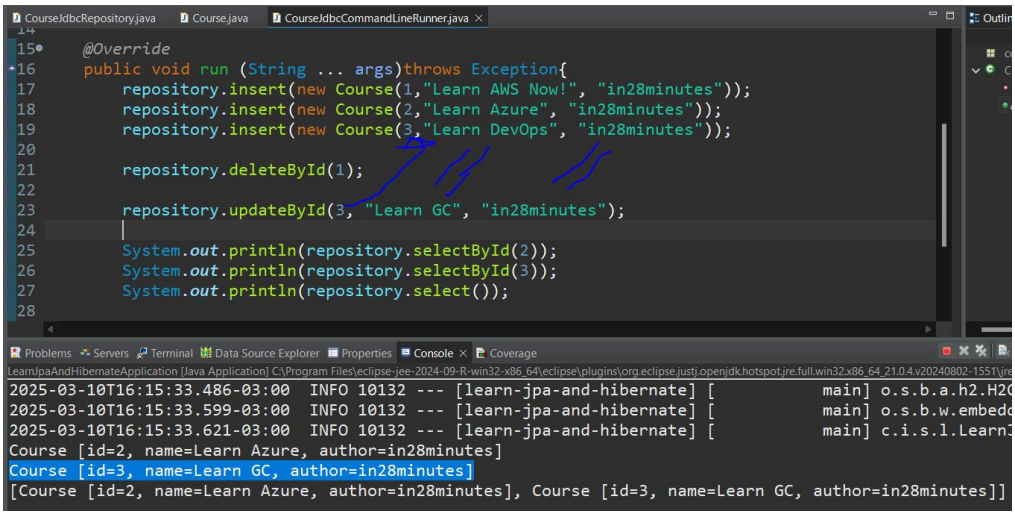
```
@Override
public void run (String ... args)throws Exception{
    repository.insert(new Course(1,"Learn AWS Now!", "in28minutes"));
    repository.insert(new Course(2,"Learn Azure", "in28minutes"));
    repository.insert(new Course(3,"Learn DevOps", "in28minutes"));

    repository.deleteById(3);
}
```



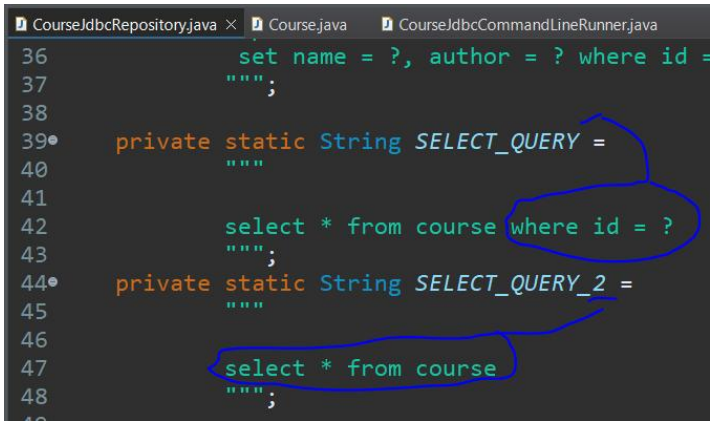
Applying the same logic, the delete query creates an sql statement of deletion, then we use a method to delete the register by id and call the method on the commandLineRunner.

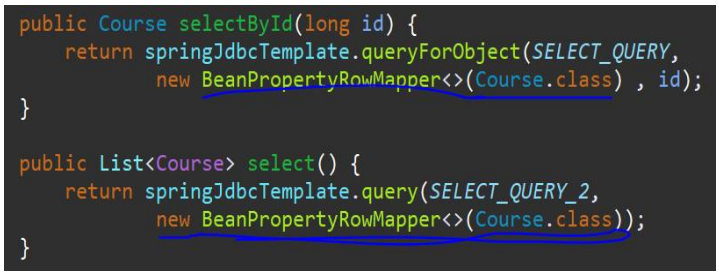
```
private static String UPDATE_QUERY =
        """

         update course
         set name = ?, author = ? where id = ?
        """;
```

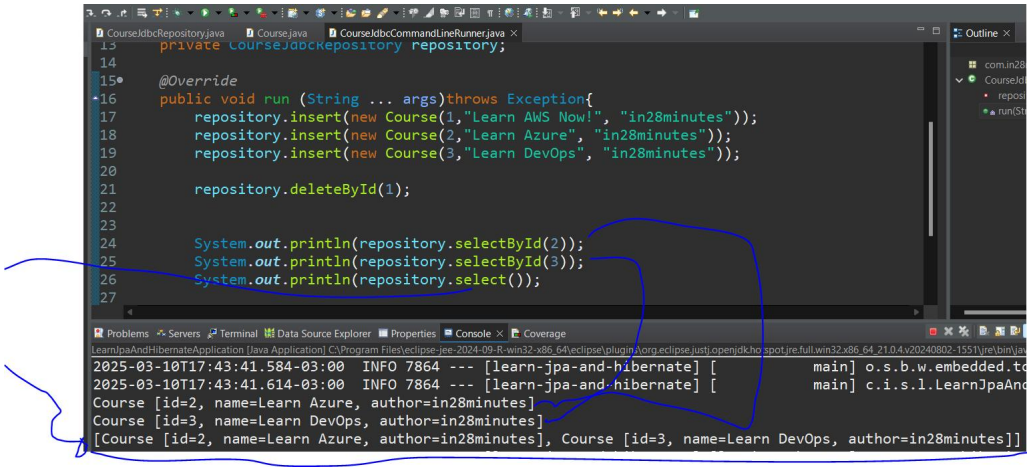
```
public void updateById(long id,String name, String author) {
    springJdbcTemplate.update(UPDATE_QUERY,
            name,author,id);
}
```



Applying the same logic, the update query creates a sql statement of update, then we create a method to update the register by id, with the values to be updated and call it on the commandLineRunner.



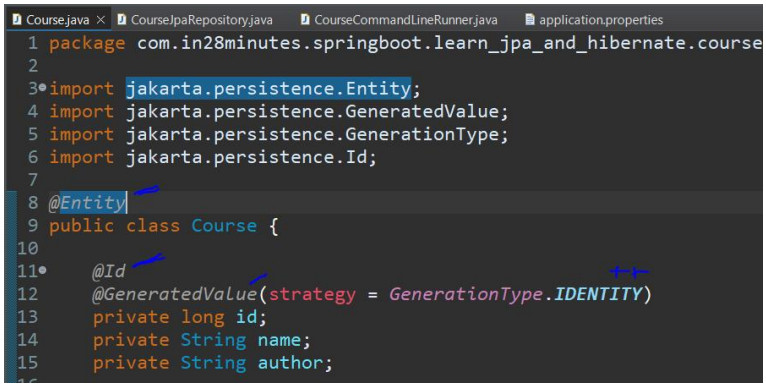




The logic is almost the same, sql statement for selection with or without id, the method to return one row or multiple rows, but for **select operation** is needed to instantiate **a new object** to shell the resultset based on the class Course for both queryForObject (retrieve one row) and query(retrieve all).

JPA (Java Persistence API)(raw)

On JPA, when creating the classes that will represent the tables on the database, it is needed to include the annotations:



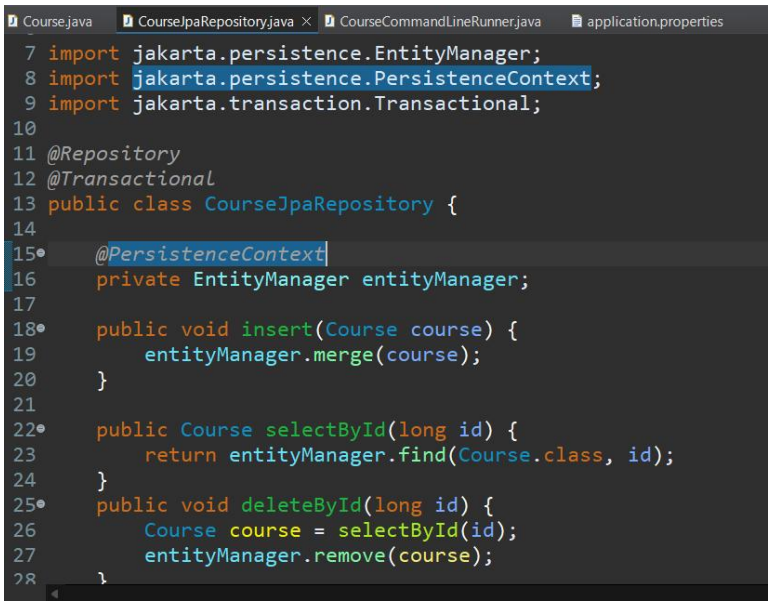
@Entity-> Map the java class as a table in the database

@Table(name="any name")-> Specify the name of the table **independently** the name of the class

@Id-> Define the primary key of the table

@GeneratedValue(strategy = GenerationType.IDENTITY)-> generate incremental ids starting by 1 and plus 1 at each new register

@Column(name="any name")-> Specify the name of the field/column of the table **independently** the name of the class attribute



@Repository -> Identify the class as the data access layer for **a specific table**

@Transactional-> To declare that the repository will make database transactions

@PersistenceContext -> When using the Entity Manager object is better to use @persistenceContext instead of @Autowired to instantiate it, to be directly connected to it and the persistence on database.

To do the operations, it is only needed to use the entityManager object and it's methods:

Merge-> to insert/update a new row

Find-> to select **a** row according to the class and id

Remove-> to delete a row from the table according to the object.

In a direct comparison with jdbc, it is not needed to write sql statements, only few java code.

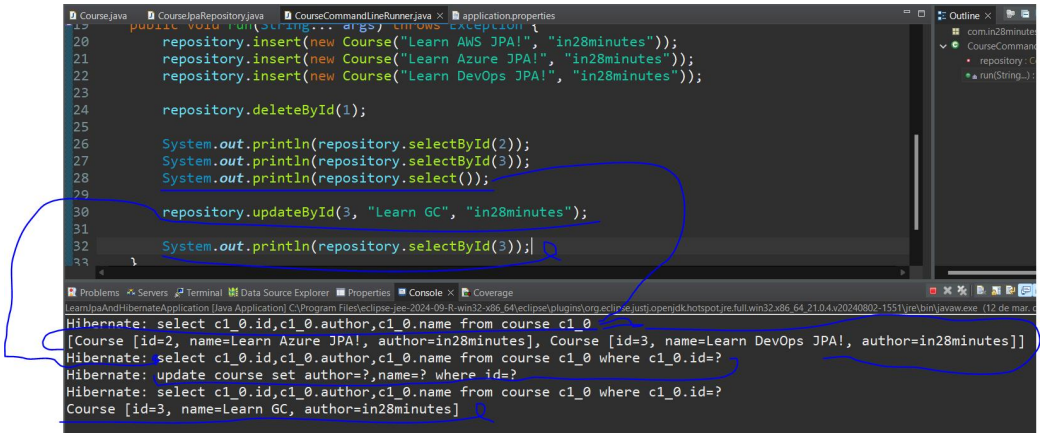
```java
@Component
public class CourseCommandLineRunner implements CommandLineRunner {

	//@Autowired
	//private CourseJdbcRepository repository;

	@Autowired
	private CourseJpaRepository repository;
```

```
spring.application.name=learn-jpa-and-hibernate
spring.h2.console.enabled=true
spring.datasource.url=jdbc:h2:mem:testdb
spring.jpa.show-sql=true
```

```
Maximum pool size: undefined/unknown
2025-03-12T12:34:48.715-03:00  INFO 12032 --- [learn-jpa-and-hibernate] [           main] o.h.e.t.j.p.i.JtaPlatformInitiator
Hibernate: drop table if exists course cascade
Hibernate: create table course (id bigint generated by default as identity, author varchar(255), name varchar(255), primary key (id
2025-03-12T12:34:48.776-03:00  INFO 12032 --- [learn-jpa-and-hibernate] [           main] j.LocalContainerEntityManagerFactoryBean
2025-03-12T12:34:49.270-03:00  WARN 12032 --- [learn-jpa-and-hibernate] [           main] JpaBaseConfiguration$JpaWebConfiguration
2025-03-12T12:34:50.018-03:00  INFO 12032 --- [learn-jpa-and-hibernate] [           main] o.s.b.a.h2.H2ConsoleAutoConfiguration
2025-03-12T12:34:50.159-03:00  INFO 12032 --- [learn-jpa-and-hibernate] [           main] o.s.b.w.embedded.tomcat.TomcatWebServer
2025-03-12T12:34:50.181-03:00  INFO 12032 --- [learn-jpa-and-hibernate] [           main] c.i.s.l.LearnJpaAndHibernateApplication
Hibernate: insert into course (author,name,id) values (?,?,default)
Hibernate: insert into course (author,name,id) values (?,?,default)
Hibernate: insert into course (author,name,id) values (?,?,default)
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0 where c1_0.id=?
Hibernate: delete from course where id=?
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0 where c1_0.id=?
```

On the property file, include spring.jpa.showl-sql=true, for display the sql code on the terminal.

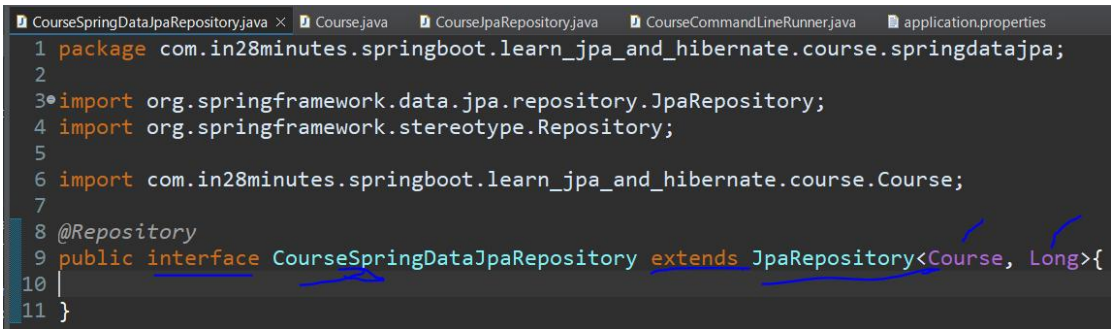
```java
    }

    public void updateById(long id, String name, String Author) {
        Course courseFound = selectById(id);
        courseFound.setName(name);
        courseFound.setAuthor(Author);
        entityManager.merge(courseFound);
    }

    public List<?> select() {
        List<?> courses = null;
        courses = entityManager.createQuery("from Course").getResultList();
        return courses;
    }
```

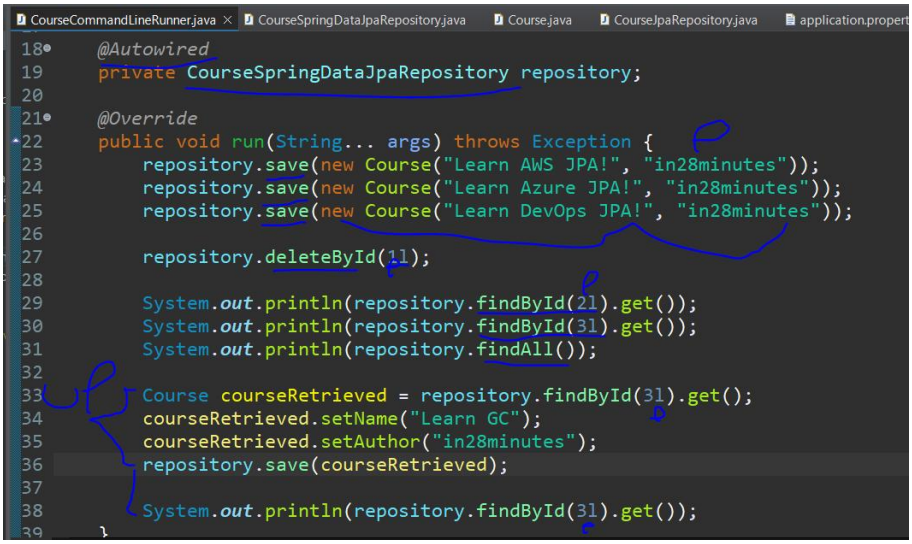


To update a row is needed to find the row by the primary key, store it in a new Course object, set new values and use the merge method with the new object. Now, for select all the rows, is needed to use the method createQuery from entityManager passing “from Course” as sql statement and get the result list as a generic type.

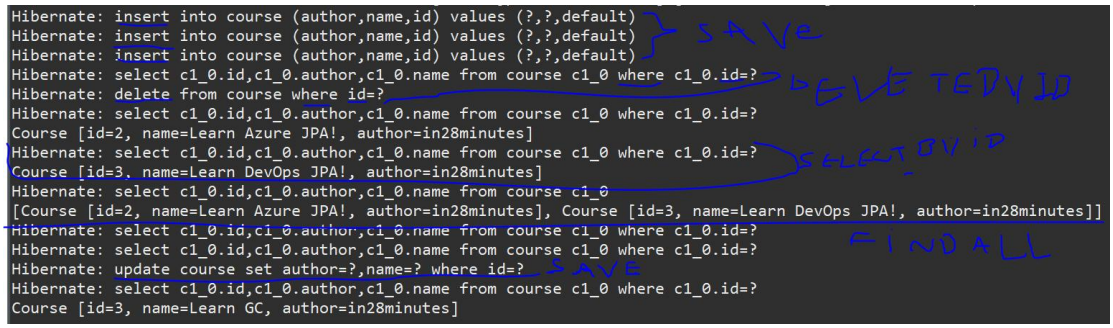
Spring **Data** JPA

On Spring Data JPA, the entityManager object and its operations are working **on background** without generating manual java code, this way, it is only needed to create an interface that will **extend** the **JpaRepository** interface passing as type <Class name, Id type>. With this interface on hands (@Autowired), the database operations/methods are just waiting for **p**arameters to be used. *3**l**-> **l** means long

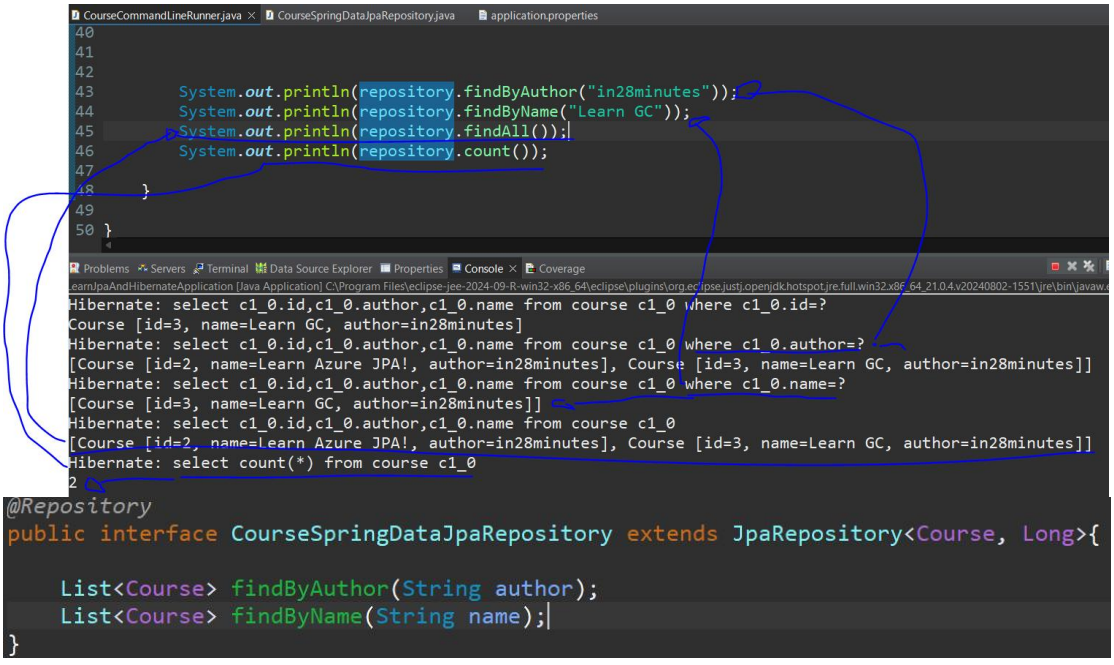




```
CourseCommandLineRunner.java | CourseSpringDataJpaRepository.java | Course.java | CourseJpaRepository.java | application.propert

18    @Autowired
19    private CourseSpringDataJpaRepository repository;
20
21    @Override
22    public void run(String... args) throws Exception {
23        repository.save(new Course("Learn AWS JPA!", "in28minutes"));
24        repository.save(new Course("Learn Azure JPA!", "in28minutes"));
25        repository.save(new Course("Learn DevOps JPA!", "in28minutes"));
26
27        repository.deleteById(11);
28
29        System.out.println(repository.findById(21).get());
30        System.out.println(repository.findById(31).get());
31        System.out.println(repository.findAll());
32
33        Course courseRetrieved = repository.findById(31).get();
34        courseRetrieved.setName("Learn GC");
35        courseRetrieved.setAuthor("in28minutes");
36        repository.save(courseRetrieved);
37
38        System.out.println(repository.findById(31).get());
39    }
```



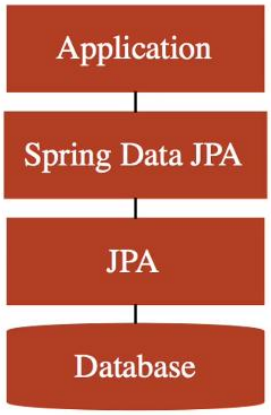
```
Hibernate: insert into course (author,name,id) values (?,?,default)
Hibernate: insert into course (author,name,id) values (?,?,default)
Hibernate: insert into course (author,name,id) values (?,?,default)
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0 where c1_0.id=?
Hibernate: delete from course where id=?
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0 where c1_0.id=?
Course [id=2, name=Learn Azure JPA!, author=in28minutes]
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0 where c1_0.id=?
Course [id=3, name=Learn DevOps JPA!, author=in28minutes]
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0
[Course [id=2, name=Learn Azure JPA!, author=in28minutes], Course [id=3, name=Learn DevOps JPA!, author=in28minutes]]
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0 where c1_0.id=?
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0 where c1_0.id=?
Hibernate: update course set author=?,name=? where id=?
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0 where c1_0.id=?
Course [id=3, name=Learn GC, author=in28minutes]
```



```
CourseCommandLineRunner.java | CourseSpringDataJpaRepository.java | application.properties

40
41
42
43        System.out.println(repository.findByAuthor("in28minutes"));
44        System.out.println(repository.findByName("Learn GC"));
45        System.out.println(repository.findAll());
46        System.out.println(repository.count());
47
48    }
49
50 }

Problems | Servers | Terminal | Data Source Explorer | Properties | Console | Coverage
LearnJpaAndHibernateApplication [Java Application] C:\Program Files\eclipse-jee-2024-09-R-win32-x86_64\eclipse\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_21.0.4.v20240802-1551\jre\bin\javaw.e
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0 where c1_0.id=?
Course [id=3, name=Learn GC, author=in28minutes]
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0 where c1_0.author=?
[Course [id=2, name=Learn Azure JPA!, author=in28minutes], Course [id=3, name=Learn GC, author=in28minutes]]
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0 where c1_0.name=?
[Course [id=3, name=Learn GC, author=in28minutes]]
Hibernate: select c1_0.id,c1_0.author,c1_0.name from course c1_0
[Course [id=2, name=Learn Azure JPA!, author=in28minutes], Course [id=3, name=Learn GC, author=in28minutes]]
Hibernate: select count(*) from course c1_0
2
```

```
@Repository
public interface CourseSpringDataJpaRepository extends JpaRepository<Course, Long>{

    List<Course> findByAuthor(String author);
    List<Course> findByName(String name);
}
```

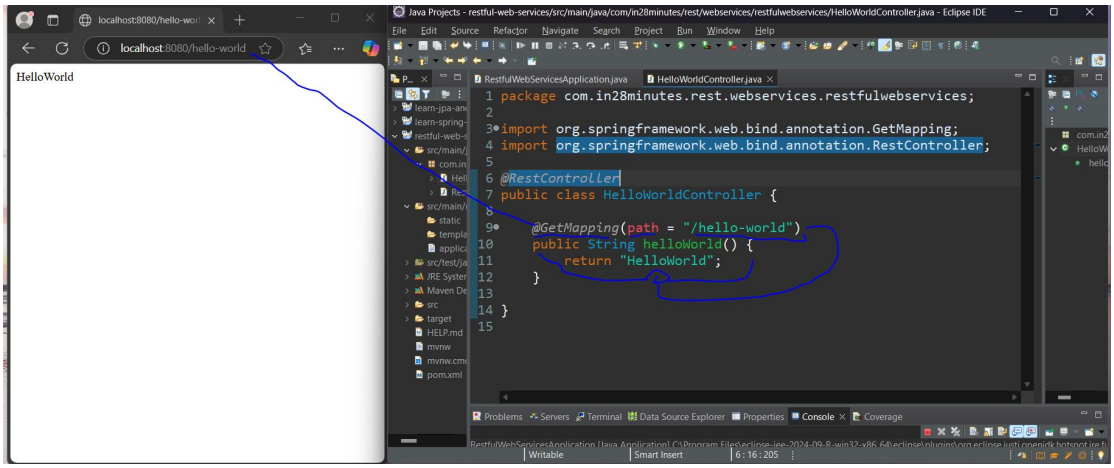
The JpaRepository also allows to define customized methods in our new interface and directly use them without worrying about the logic, for example find some course by Author, Name.



As a result of the structure, the app uses the Spring Data JPA that uses JPA in background to access the database.

So to finalize the appendix, whats is the difference between JPA and Hibernate? JPA is the **specification** in how to map the entities, attributes and manage the entities (entity manager, crud) and hibernate is **one of the possible implementations** of the JPA. So, it's important to always import from the **specification** when using the @Annotations, because the specification hardly changes, now the implementation changes more easily.

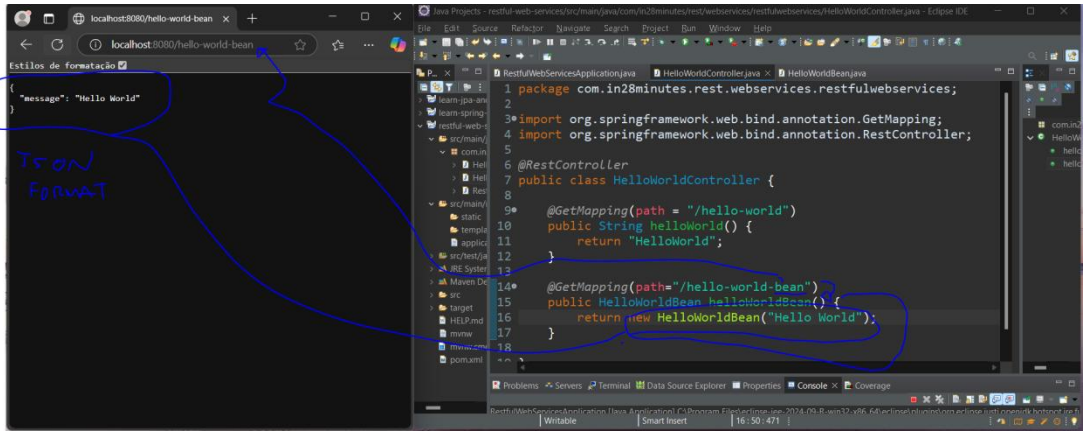
Building a sample spring boot restful API



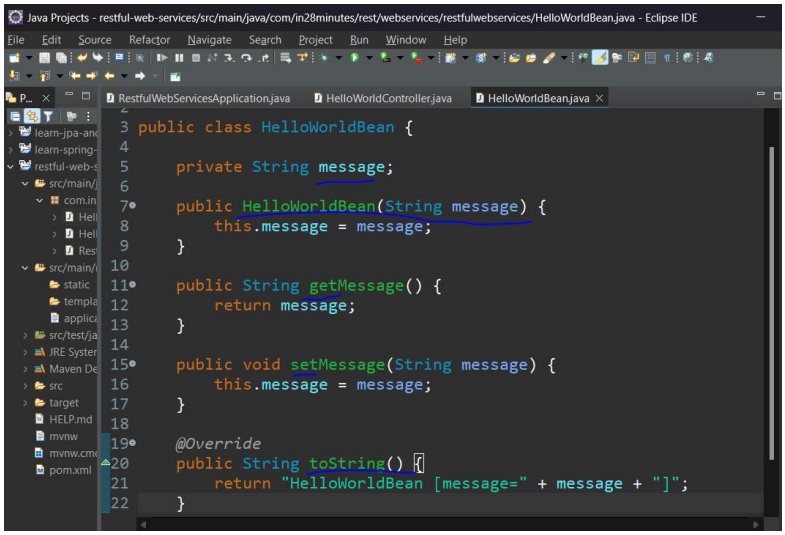
After generating the project with spring web, devtools, h2 and spring data jpa dependencies, it is needed to create a **controller** class that will have the responsibility to exposure the endpoints/uris and serve the http requests, containing a method to return the String Hello World. As important, the annotations will make this happen:

@RestController -> it identifies the class as a restful controller component to expose endpoints/uris and serve the http requests.

@GetMapping(path = “/hello-world”) -> it maps all the GET requests with the path(uri) “/hello-world” and trigger the method below the annotation.



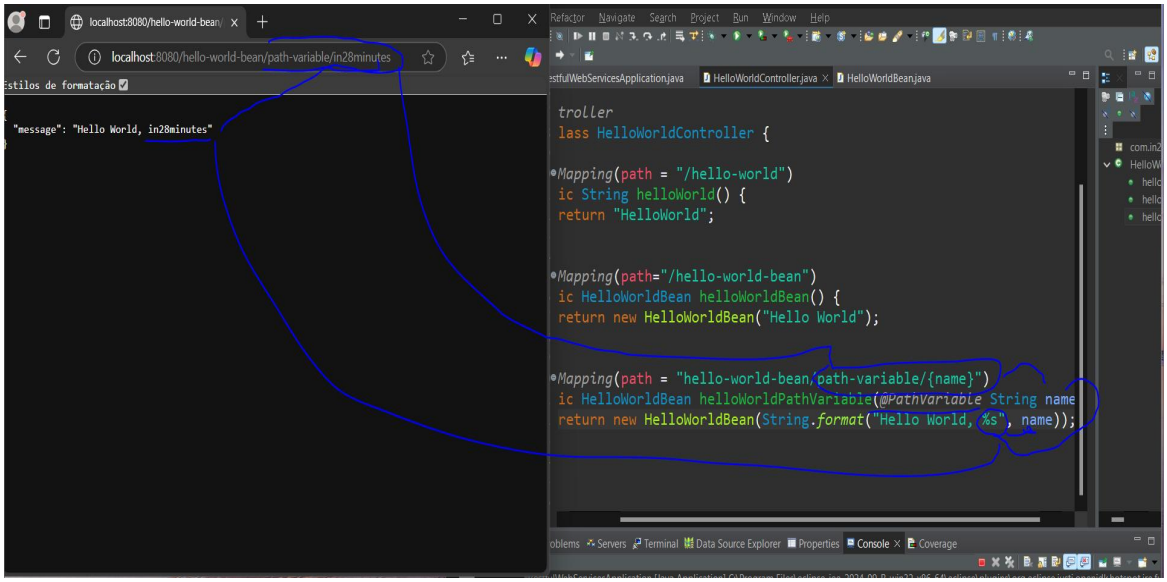
In the real world, many times, what the request expects is a json object, for that it is needed to create a method mapped as a get request that will return this bean(object). Based on it, it is created the method helloWorldBean with the annotation @GetMapping(path=”hello-world-bean”) returning this bean in json format.



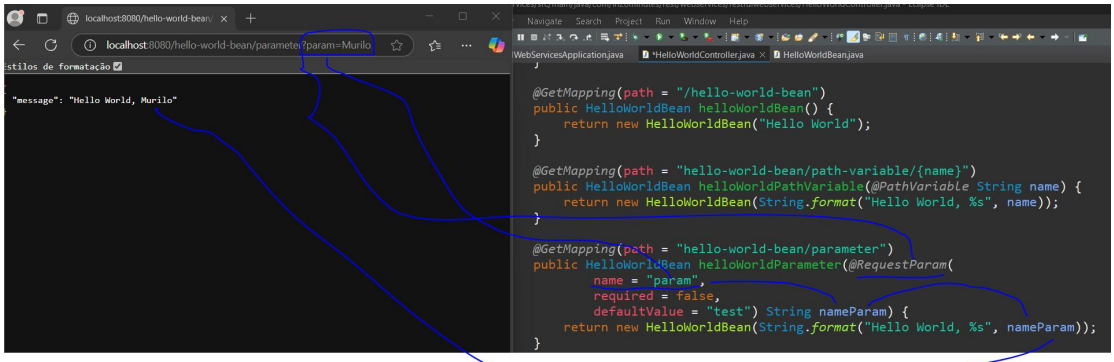
The Bean structure is a simple class with the attribute message, a constructor receiving this message, getter, setter and toString methods.

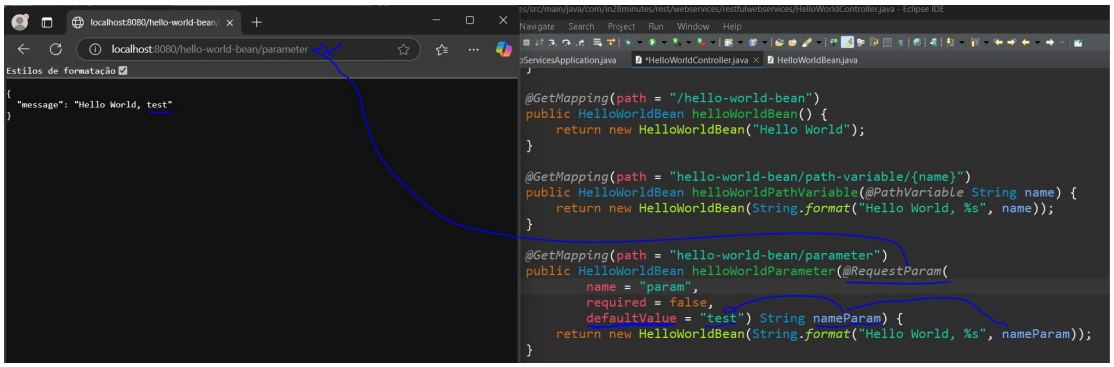
Some important points, every request is mapped to a specific controller that will trigger one specific method to return a response. The **dispatch servlet** that comes from sping mvc is the front controller that will handle **all the requests of the app** and point out which is the controller responsible for serving that specifically request to discover the method responsible for the operation. Then, our @RestController will take care of the response. Finally, the HttpMessageConverter and Jackson2Object are the jars responsible for converting a bean to a JSON and a JSON to a bean.

```
## Social Media Application Resource Mappings

### User -> Posts

- Retrieve all Users      - GET  /users
- Create a User           - POST /users
- Retrieve one User       - GET  /users/{id} -> /users/1
- Delete a User           - DELETE /users/{id} -> /users/1

- Retrieve all posts for a User - GET /users/{id}/posts
- Create a posts for a User - POST /users/{id}/posts
- Retrieve details of a post - GET /users/{id}/posts/{post_id}
```



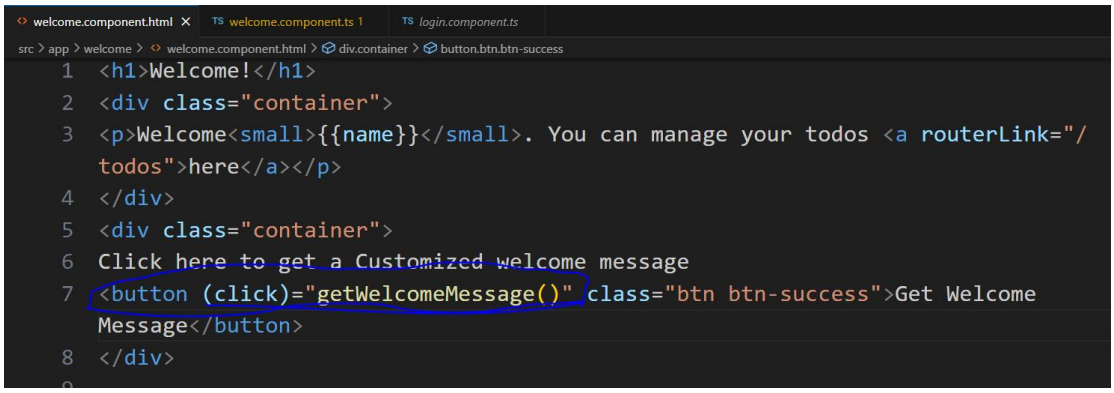
To finalize the section, the user can pass some value through the url (domain+uri), as in the information received **through the url** is identified as a **{variable}** that can be used **on the mapping.** the method, this value is known as **@PathVariable** the **name** of the user can be used to build the message on the instance of the bean-> string.format("Hello World, %s", **name**);. *%s-> kind of data of the variable, s for String, l for Long, i for Integer and so on.



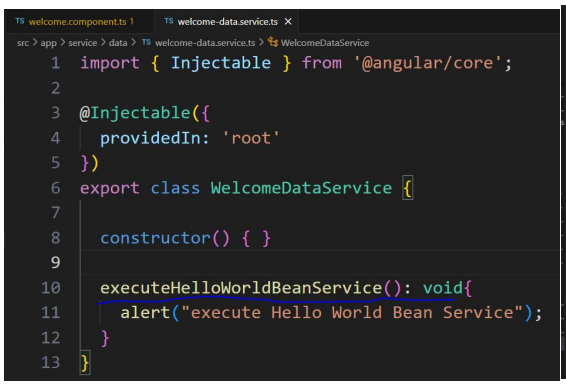


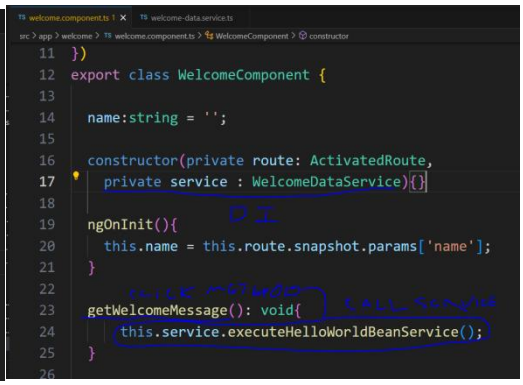
Another way to receive information through the url is using a parameter, in this case the request parameter, for that it is needed to use the **@RequestParam annotation** on the method signature and give details to it like name = “param”, required = true/false and in case of null use the field defaultValue = “test” to save some information.

Angular and Spring Boot Integration

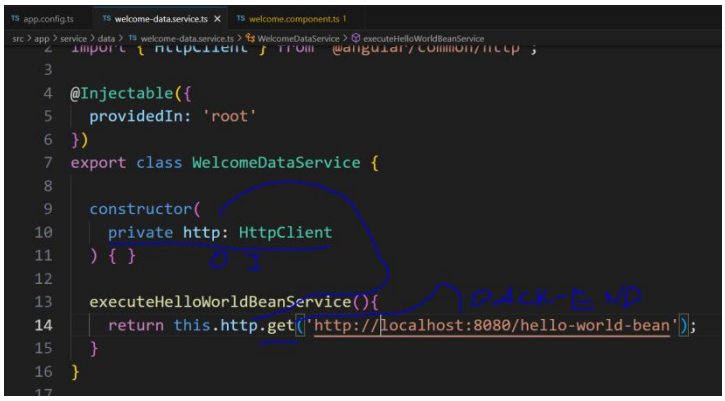


This integration starts with a test, lets say that the angular app needs to get a customized message from the back-end, creating a button on the welcome.component.**html** containing a click event biding calling the method “getWelcomeMessage()” that is present on the .ts file.





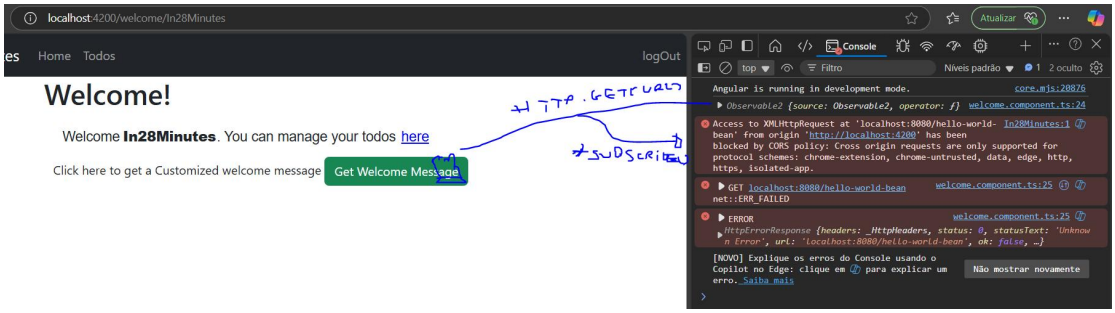
To focus this back-end integration logic, it is created the service WelcomeDataService with the method executeHelloWorldBeanService. For using this service, it is need that it be ‘used’ through the dependency injection on the constructor of the welcome component and inside the method related to the (click) event to call the execute service method.



For testing the connection in fact, it is needed to make the dependency injection of the httpClient class on the service, when clicking on the button it will call the service method, this method will use this reference to make the get request to the back-end (“localhost:8080/hello-world-bean”), for now, it will return an Observable object, because actually, it is not making the call exactly.

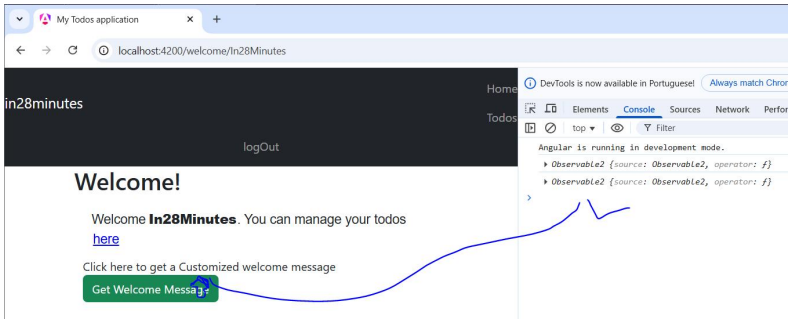
```ts
import { FormsModule } from '@angular/forms';
import { BrowserModule } from '@angular/platform-browser';
import { ApplicationConfig, importProvidersFrom } from '@angular/core';
import { provideRouter } from '@angular/router';

import { routes } from './app.routes';

import { provideHttpClient, withInterceptorsFromDi } from '@angular/common/http';

export const appConfig: ApplicationConfig = {
  providers: [
    importProvidersFrom(BrowserModule,FormsModule),
    provideRouter(routes),
    provideHttpClient(withInterceptorsFromDi())
  ]
};
```

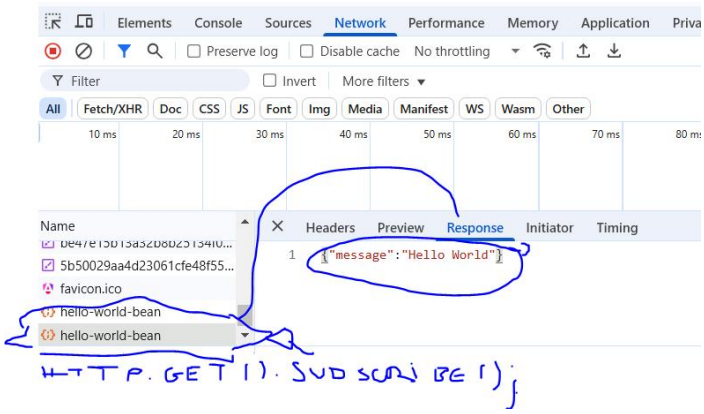
For using this httpClient Module, it is needed to import the modules FormsModule, BrowserModule, importProvidersFrom, provideHttpClient, withInterceptorsFromDi. Then, use the importProvidersFrom passing the BrowserModule and FormsModule as parameters and using the provideHttpClient passing the withInterceptorsFromDi as parameter.



For really making the request to back-end server, it is needed after calling the service, return an observable, concatenating it with .subscribe() for literally connect both sides to obtain the expected object.

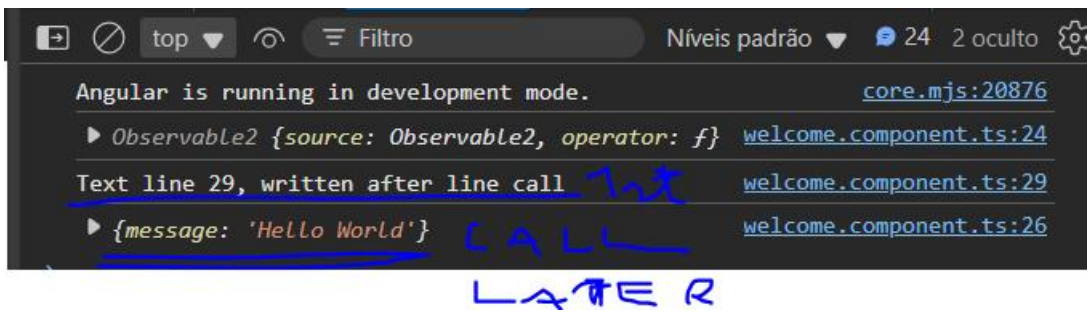
```java
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
@CrossOrigin(origins="http://localhost:4200")
public class HelloWorldController {

    @GetMapping(path = "/hello-world")
    public String helloWorld() {
        return "HelloWorld";
    }

    @GetMapping(path = "/hello-world-bean")
    public HelloWorldBean helloWorldBean() {
        return new HelloWorldBean("Hello World");
    }
```





But as security feature, the spring boot apps are configured to **block any external call**, this is called **CORS policy**, to allow this connection, it is needed to include on the controller class the annotation @CrossOrigin(origins=”specific front-end allowed to make call to this back-end server”).

```
export class WelcomeComponent {
  }

  getWelcomeMessage(){
      console.log(this.service.executeHelloWorldBeanService());
      this.service.executeHelloWorldBeanService().subscribe(
          response => console.log(response)
      );

      console.log("Text line 29, written after line call")
  }
}
```



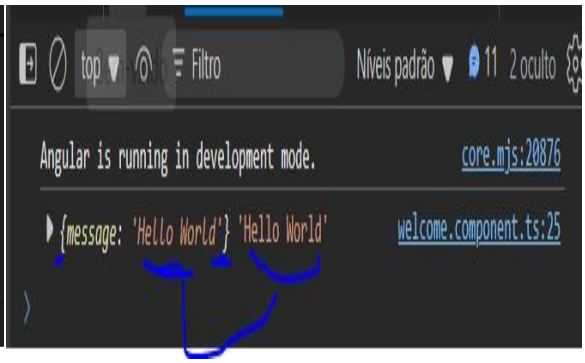
The call made by subscribe method is **asynchronous**, as in, while the call doesn't receive the response the rest of the lines can be executed (faster).

```
export class HelloWorldBean {
    constructor(public message: string){}
}

@Injectable({
  providedIn: 'root'
})
export class WelcomeDataService {

  constructor(
    private http: HttpClient
  ) { }

  executeHelloWorldBeanService(){
    return this.http.get<HelloWorldBean>('http://localhost:8080/hello-world-bean');
  }
```

For casting the response object into a new object on angular, it is needed to create a class containing the message attribute and pass it as a generic type<>, for that the object be displayed and usable on angular.

```
export class WelcomeComponent {

  ngOnInit(){
    this.name = this.route.snapshot.params['name'];
  }

  getWelcomeMessage(){
      this.service.executeHelloWorldBeanService().subscribe(
          response => console.log(response, response.message)
      );
```

VALUE



Having the shell object, its content can be usable and manipulated, for instance, the whole object, value of its attributes and so on.

```
export class WelcomeComponent {
  name:string = '';
  customizedWelcomeMessage:string= '';

  constructor(private route: ActivatedRoute,
    private service : WelcomeDataService){}

  ngOnInit(){
    this.name = this.route.snapshot.params['name'];
  }

  getWelcomeMessage(){
      this.service.executeHelloWorldBeanService().subscribe(
          response => this.customizedWelcomeMessage = response.message
      );
```

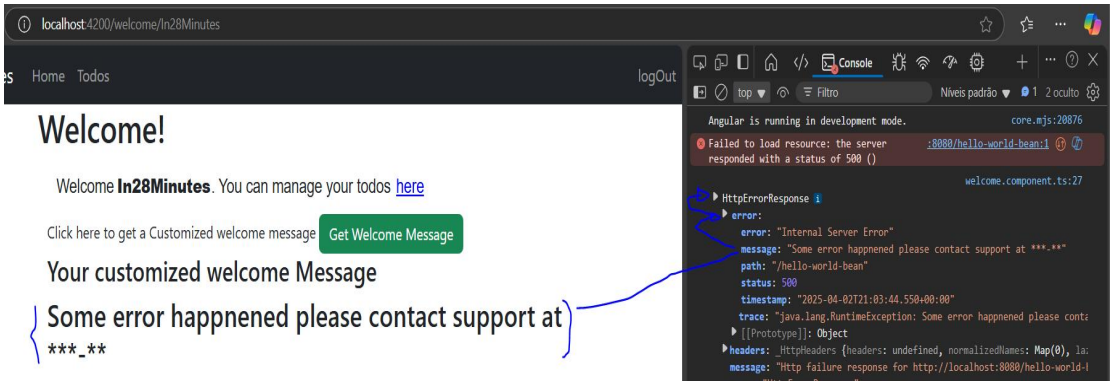
```
welcome > welcome.component.html > div.container
<div class="container">
todos">here</a></p>
</div>
<div class="container">
Click here to get a Customized welcome message
<button (click)="getWelcomeMessage()" class="btn btn-succe
Message</button>
</div>
<div class="container" *ngIf="customizedWelcomeMessage">
<h3>Your customized welcome Message</h3>
<h2>{{customizedWelcomeMessage}}</h2>
</div>
```

With this object in hands, it is interesting to show it inside the welcome template. It is needed to use the new customized message property, on the lambda back-end response to receive the message from the object response and display it through {{interpolation}} since it is filled in(*ngIf).

(back-end)

```
@RestController
@CrossOrigin(origins="http://localhost:4200")
public class HelloWorldController {

    @GetMapping(path = "/hello-world")
    public String helloWorld() {
        return "HelloWorld";
    }

    @GetMapping(path = "/hello-world-bean")
    public HelloWorldBean helloWorldBean() {
        //return new HelloWorldBean("Hello World");
        throw new RuntimeException("Some error happnened please contact support at ***-**");
    }
```

```
src > app > welcome > welcome.component.ts > WelcomeComponent > getWelcomeMessage > subscribe() callback
12  export class WelcomeComponent {
19
20    ngOnInit(){
21      this.name = this.route.snapshot.params['name'];
22    }
23
24    getWelcomeMessage(){
25        this.service.executeHelloWorldBeanService().subscribe(
26            response => this.customizedWelcomeMessage = response.message,
27            error =>  { this.customizedWelcomeMessage = error.error.message,
28                        console.log(error)}
29        );
```



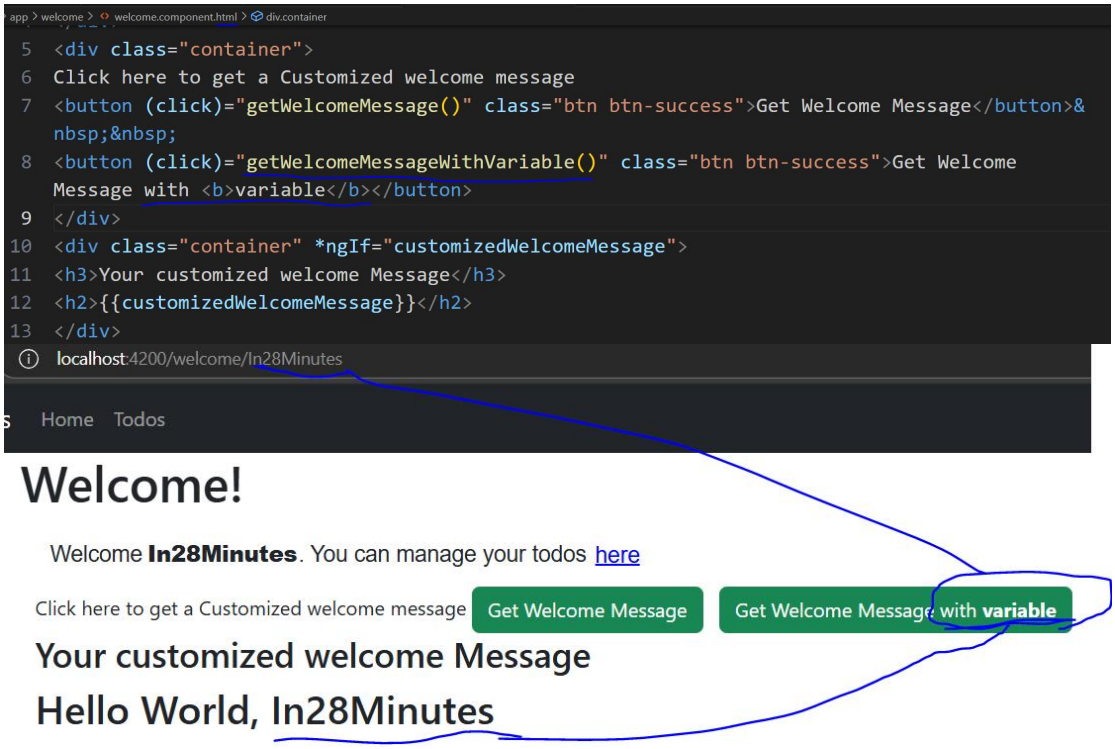
The same way that there is a response object for success, there is an **error object** for failure on angular. In this case, it's possible to reuse the same customizedWelcomeMessage property to receive the message from back-end and display it on the {{template}}. On the back-end side, it is needed to **throw** this error (new RuntimeException("")).

```
app.config.ts | welcome-data.service.ts | welcome.component.ts | welcome.component.html
src > app > service > data > welcome-data.service.ts > WelcomeDataService
11  export class WelcomeDataService {
17    executeHelloWorldBeanService(){
19    }
20
21    executeHelloWorldBeanServiceWithVariable(name:string){
22        return this.http.get<HelloWorldBean>(`http://localhost:8080/hello-world-bean/
          path-variable/${name}`);
23    }
24  }
```

Going to the next call, on the service is created one more method that will receive a parameter on the signature to include it inside the url(`using a backtick instead of quotes`) of the call (${name}).

```
export class WelcomeComponent {
  }

  getWelcomeMessageWithVariable(){
    this.service.executeHelloWorldBeanServiceWithVariable(this.name).subscribe(
      response => this.customizedWelcomeMessage = response.message,
      error => this.customizedWelcomeMessage = error.error.message
    );
  }
```

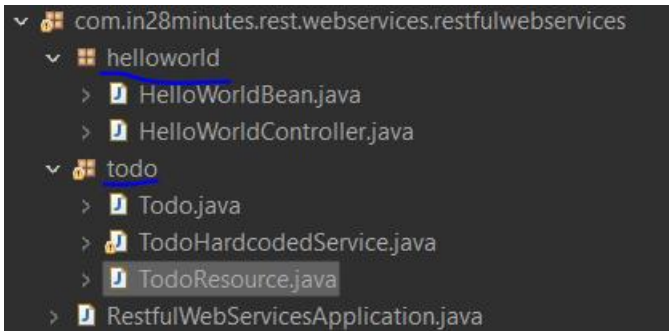
On the component.ts, the property name is passed **as a variable** on the service call new method.



On the template, one more button is created calling this new method.

With this structure on hands, the **to do** operations can be done following the below idea for the controller:

```
Retrieve all Todos for A User
GET /users/{user_name}/todos

Delete a Todo of a User
DELETE /users/{user_name}/todos/{todo_id}

Edit/Update a Todo
PUT /users/{user_name}/todos/{todo_id}

Create a new Todo
POST /users/{user_name}/todos/
```



Separate the HelloWorld content in a new package, create another one for the todo content.

```java
@RestController
public class TodoResource {

    @Autowired
    private TodoHardcodedService todoService;

    @GetMapping(path="/users/{username}/todos")
    public List<Todo> getAllTodos(@PathVariable String username){
        return todoService.findAll();
    }

}
```

On the todo package, create a controller that will return a list<Todo> in a method as a @GetMapping request containing the path variable, using the service to be created later to return it.

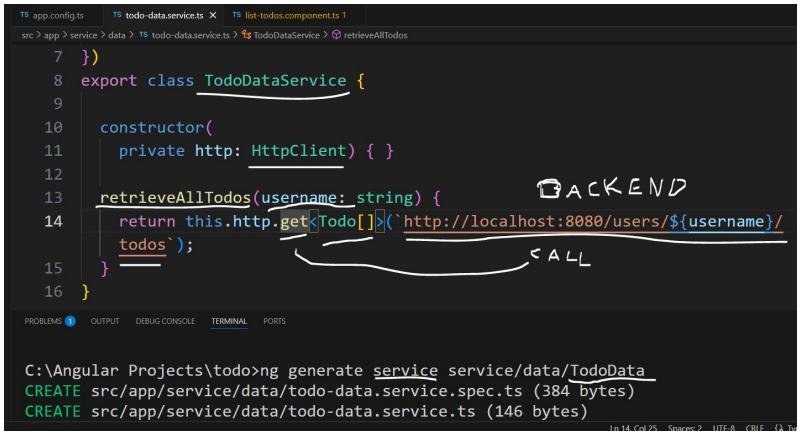
```java
public class Todo {

    private long id;
    private String username;
    private String description;
    private Date targetDate;
    private boolean isDone;

    public Todo(long id, String username, String description,
                Date targetDate, boolean isDone) {
        this.id = id;
        this.username = username;
        this.description = description;
        this.targetDate = targetDate;
        this.isDone = isDone;
    }

    public long getId() {
        return id;
    }
```

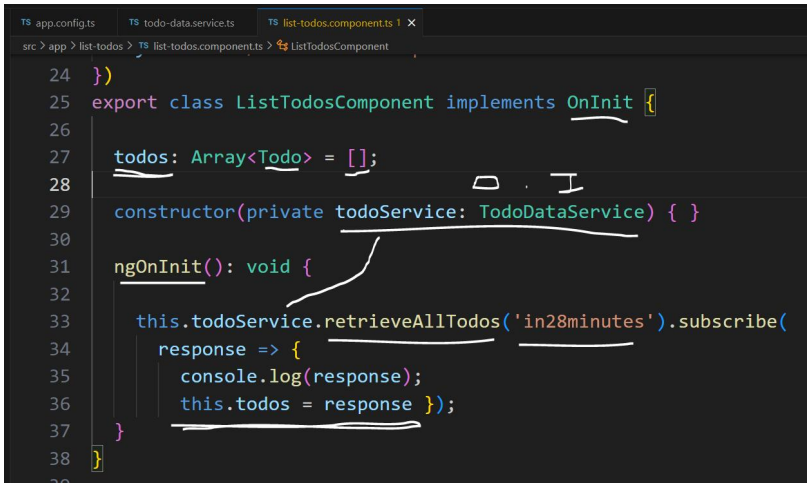
Then, create the relational object Todo with id,username,description,targetDate and isDone as a boolean state with getters,setters and constructor.

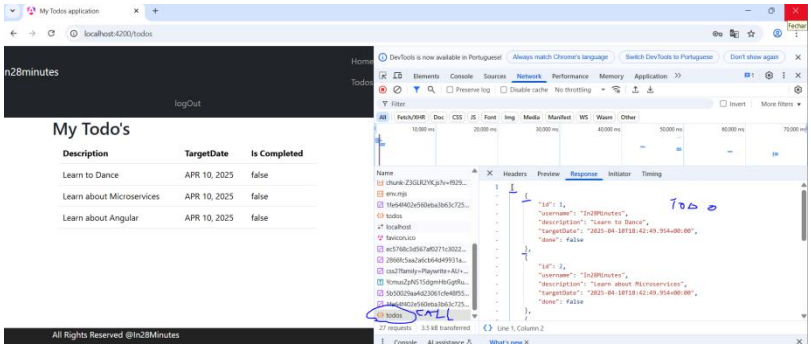
```java
@Service
public class TodoHardcodedService {

    private static List<Todo> todos = new ArrayList();
    private static int idCounter = 0;

    static {
        todos.add(new Todo(++idCounter,"In28Minutes","Learn to Dance", new Date(),false));
        todos.add(new Todo(++idCounter,"In28Minutes","Learn about Microservices", new Date(),false));
        todos.add(new Todo(++idCounter,"In28Minutes","Learn about Angular", new Date(),false));
    }

    public List<Todo> findAll(){
        return todos;
    }
}
```

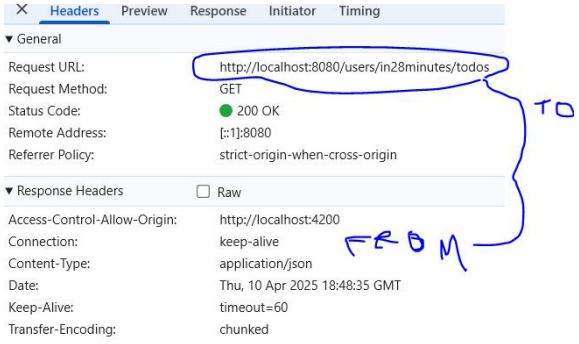
The business rule is focused on a new hardcoded service with the list<todo> as static, an idCounter. Create the static declaring and put on the {} the addition of three todos, pre-implementing the idCounter in all, learning to dance, learn about microservices, learn about Angular, and return this filled in list in a findAll method.

This is the result from the backend, a list of objects of the kind Todo.



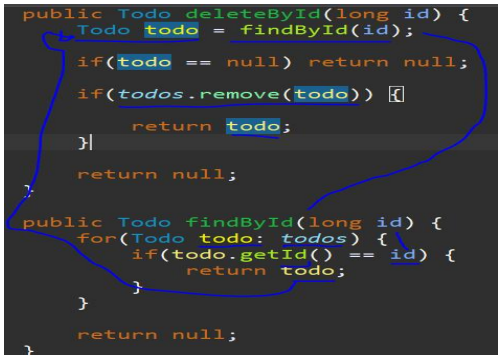




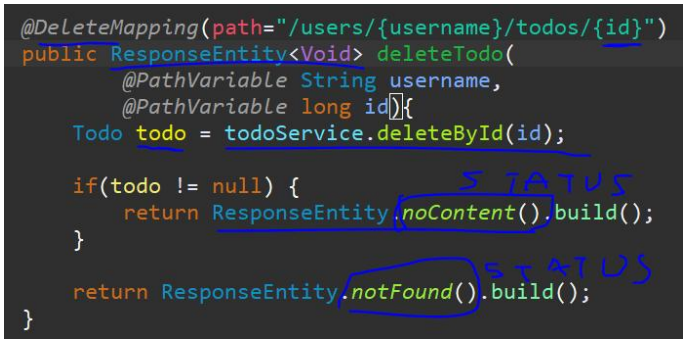


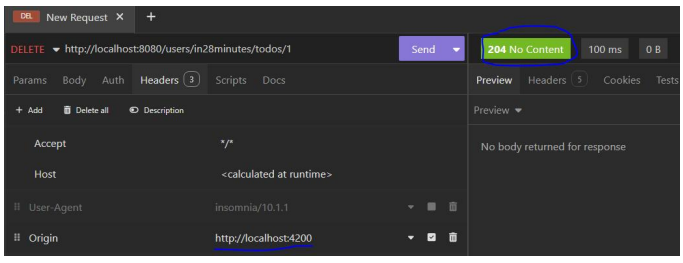
With the back-end prepared to **retrieve a list of todos**, now it's time to prepare the front-end, for that, it is generated one more service for making the todo data calls and for the list-todos.component using it, initializing the todo's list as empty, making the dependency injection of the service to call the

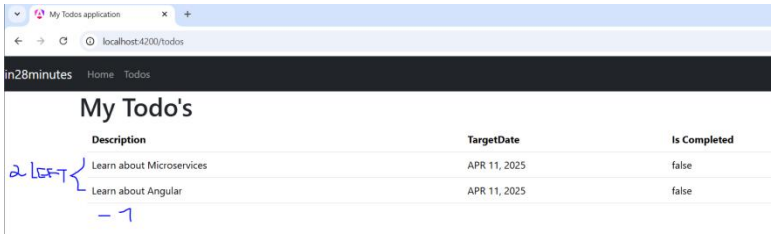
method to fill in the list . As a reminder, the todoResource class also needs to have the @CrossOrigin to allow the call from the front-end.



The next step is to simulate the deletion of a todo, for that, on the service back-end it is needed to create a method for deletion by Id, inside it create a reference of a todo that will receive the return of another method for finding the todo by id that will sweep the list to check a compatible id for it, then once returned verify if it is null, if yes, return null else verify if the todo found have been removed from the list, if true return this todo object removed else return null.







Then, for using this service, on the todoResource, create a @DeleteMapping request including the username and the id, under it, create the method with a

Response Entity<void> response, inside it, create the reference that will receive the removed todo and call the deletion method passing the id, verify if this todo is not null to return **the status no content** from Response Entity object else return **the status not found**.

```
src > app > service > data > todo-data.service.ts > TodoDataService > retrieveAllTodos
  8  export class TodoDataService {
 16
 17    deleteTodo(username:string,id:number){
 18      return this.http.delete(`http://localhost:8080/users/${username}/todos/${id}
         `);
 19    }
 20  }
```

The back-end part is prepared, now on the front-end, it is needed to create the method deleteTodo receiving the username and id to make the delete request to the back-end passing the variables.

```
export class ListTodosComponent implements OnInit {
  todos: Array<Todo> = [];

  message:string = '';

  constructor(private todoService: TodoDataService) { }

  ngOnInit(): void {
    this.refreshTodos();
  }

  refreshTodos(): void {
    this.todoService.retrieveAllTodos('in28minutes').subscribe(
      response => {
        console.log(response);
        this.todos = response });
```

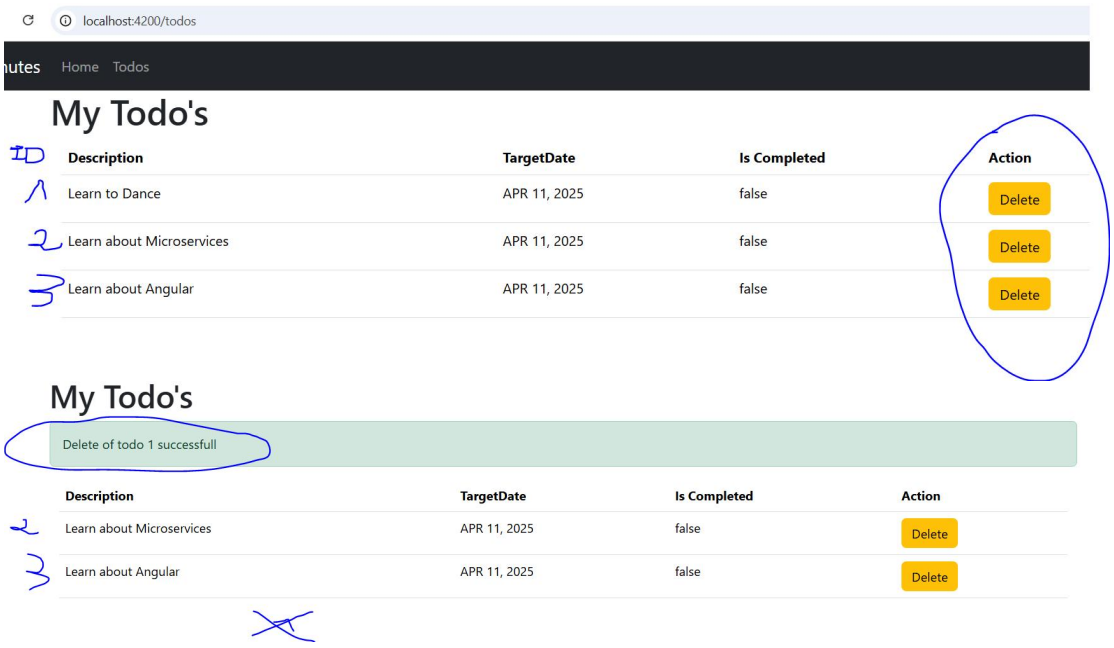
```
deleteTodo(id:number): void {
  this.todoService.deleteTodo('in28minutes',id).subscribe(
    response => {
      this.message = `Delete of todo ${id} successfull`;
      this.refreshTodos();
    }
  )
}
```

On the list todo component, a method of the same name is created receiving the id to call the delete request. On the success response, use the property message to receive the sentence, inform that the todo with the id passed was removed and update the list of todos without the removed one.

```
<tbody>
    <!--for (Todo todo : todos)
(for each element todo inside todos)-->
    <tr *ngFor="let todo of todos">
        <td>{{todo.description}}</td>
        <!-- | means convert the previous object to this format-->
        <td>{{todo.targetDate | date | uppercase}}</td>
        <td>{{todo.done}}</td>
        <td><button (click)="deleteTodo(todo.id)" class="btn btn-warning">Delete</
        button></td>
    </tr>
</tbody>
```

```
<div class="alert alert-success" *ngIf='message'>{{message}}</div>
```

On the template, it is needed to create the button inside a new <td> with the click event calling the new function passing the id of the current todo of todos. On the top of the page, it is created a div to exposure the message if it is filled in.



This way, all the functionality is accessible through the button delete.

```
<div class="container">
    <table class="table">
        <tbody>
            <tr *ngFor="let todo of todos">
                <td>{{todo.description}}</td>
                <!-- | means convert the previous object to this format-->
                <td>{{todo.targetDate | date | uppercase}}</td>
                <td>{{todo.done}}</td>
                <td><button (click)="updateTodo(todo.id)" class="btn btn-success">Update</
                button></td>
                <td><button (click)="deleteTodo(todo.id)" class="btn btn-warning">Delete</
                button></td>
            </tr>
        </tbody>
```

```
src > app > list-todos > TS list-todos.component.ts > ListTodosComponent > updateTodo
26  export class ListTodosComponent implements
47    deleteTodo(id:number): void {
54    }
55
56    updateTodo(id:number): void {
57      this.router.navigate(['/todos',id]);
58    }
59  }
```

```
C:\Angular Projects\todo>ng generate component Todo --skip-import --standalone
CREATE src/app/todo/todo.component.html (20 bytes)
CREATE src/app/todo/todo.component.spec.ts (601 bytes)
CREATE src/app/todo/todo.component.spec.ts (601 bytes)
CREATE src/app/todo/todo.component.ts (217 bytes)
CREATE src/app/todo/todo.component.css (0 bytes)
```

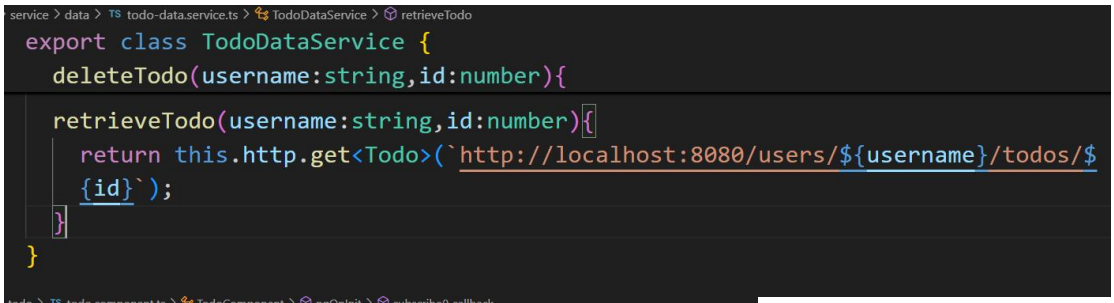
```
export const routes: Routes = [
    {path: 'login', component: LoginComponent},
    {path: 'welcome/:name', component: WelcomeComponent, canActivate:[RouteGuardService]},
    {path: 'todos', component: ListTodosComponent, canActivate:[RouteGuardService]},
    {path: 'logout', component: LogoutComponent, canActivate:[RouteGuardService]},
    {path: 'todos/:id', component: TodoComponent, canActivate:[RouteGuardService]},
    {path: '**', component: ErrorComponent}

];
```

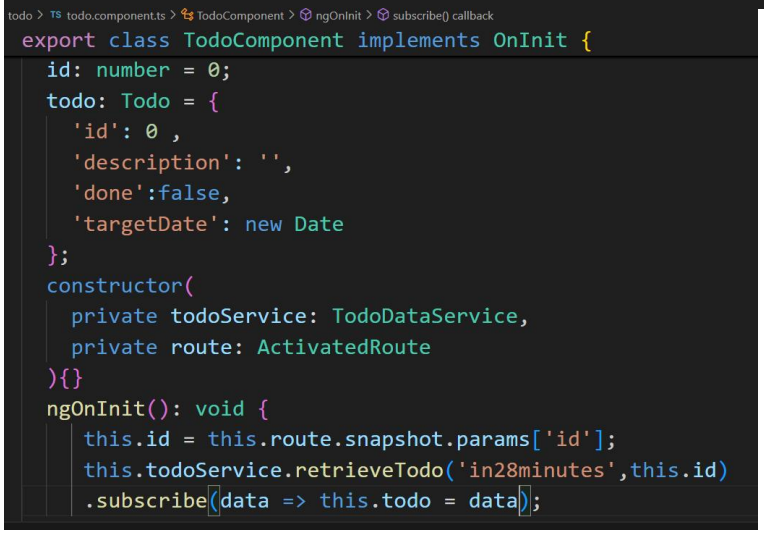
Going ahead, it's time of updating the todo, for that one more button is created to redirect the user to a new component called Todo passing the id of the selected todo. In summary, create this button (template) with the click event calling the method to redirect to this new component passing the id (.ts). This way, the component is inserted in the route list.

```
TS app.config.ts   todo.component.html ×
src > app > todo > todo.component.html > br
1  <h1>Todo</h1>
2  <br/>
3  <div class="container">
4        <fieldset class="form-group">
5           <label for="description">Description</label>
6           <input type="text" class="form-control" id="description"
            name="description" required="required">
7        </fieldset>
8        <br/>
9        <fieldset class="form-group">
10          <label for="targetDate">Target Date</label>
11          <input type="date" class="form-control" id="targetDate"
            name="targetDate" required="required">
12       </fieldset>
13       <br/>
14       <button class="btn btn-success" (click)="saveTodo()">Save</button>
```

On the todo template, create the title todo, fieldsets, labels, required inputs for text and date, and a button to trigger the method to call the back-end service.

On the back-end, it is created one route to retrieve **one** todo receiving the username and id as variables on the path and method. The search is done sweeping the todos list and checking if the current id is equal to the one sent by parameter, once found, it is returned to the front-end.





```
<br/>
<div class="container">
    <fieldset class="form-group">
      <label for="description">Description</label>
      <input type="text" [(ngModel)]="todo.description" class="form-control"
      id="description" name="description" required="required"/>
    </fieldset>
    <br/>
    <fieldset class="form-group">
      <label for="targetDate">Target Date</label>
      <input type="date" [(ngModel)]="todo.targetDate" class="form-control"
      id="targetDate" name="targetDate" required="required">
    </fieldset>
    <br/>
    <button class="btn btn-success" (click)="saveTodo()">Save</button>
```

On the front-end side, it is needed to create the call in the service receiving **a** <Todo> as type, on the todo component .ts is created the id and todo properties to store the same ones by route parameter and data obtained by the service call and finally on the template is used the directive [(ngModel)] to retrieve and update the fields description and date of the property todo.

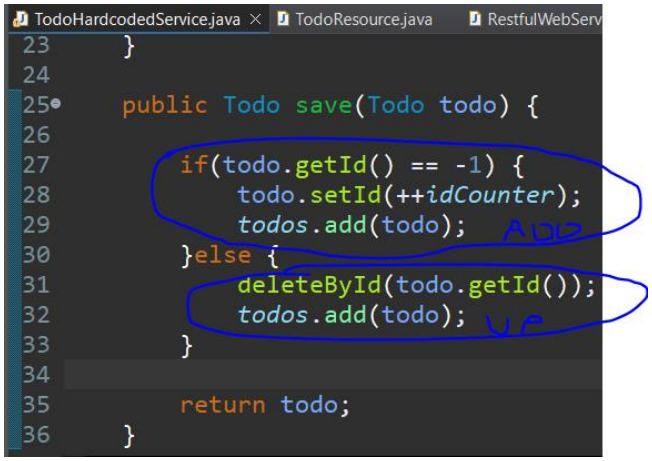
```
todo.component.html | todo-data.service.ts | todo.component.ts
do > todo.component.html > div.container > fieldset.form-group > input#targetDate.form-control
<div class="container">
    <fieldset class="form-group">
      <label for="targetDate">Target Date</label>
      <input type="date"
      [ngModel]="todo.targetDate | date: 'yyyy-MM-dd'"
      (ngModelChange)="todo.targetDate = $event"
      class="form-control" id="targetDate" name="targetDate"
      required="required">
    </fieldset>
```

But the date was not being formatted as year,month and day only, to fix the problem of the date to be in another format, it is needed to segregate the ngModel in two:

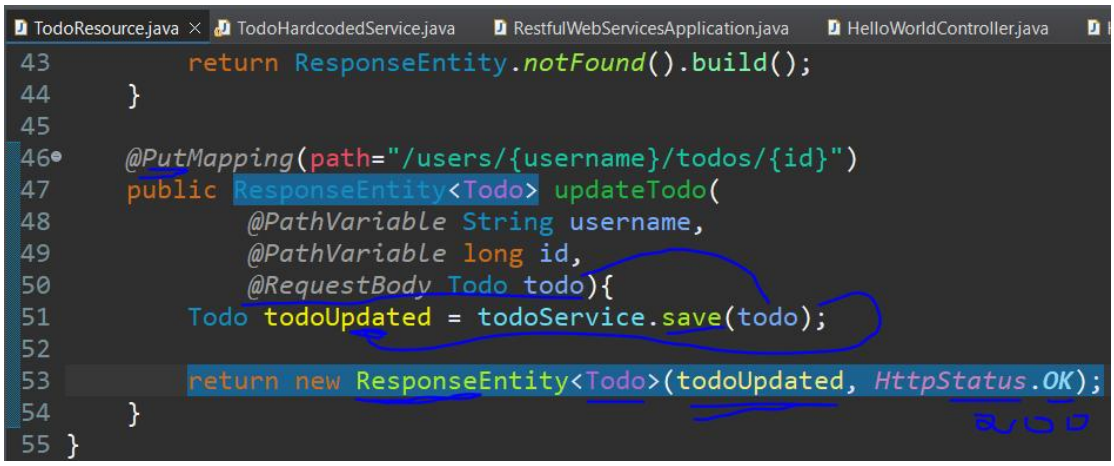
[ngModel] = ”property | date: ‘date format wished for’ ”

| means is formatted to

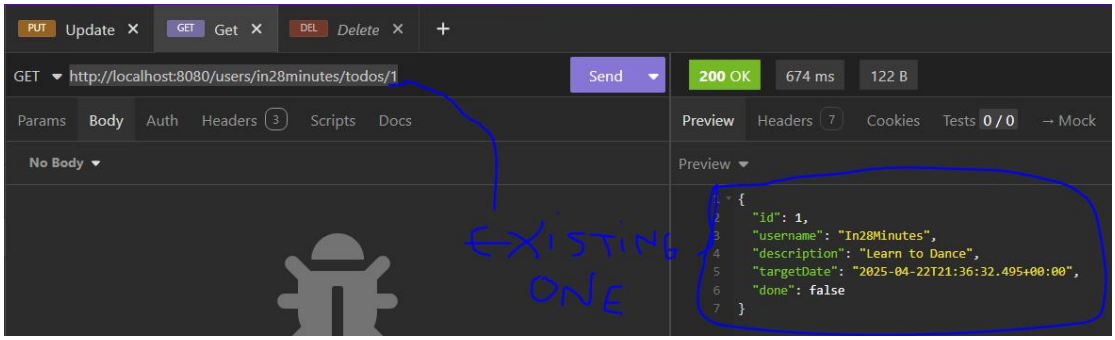
(event binding for every change of value defined by the user) = “property = $newValue”

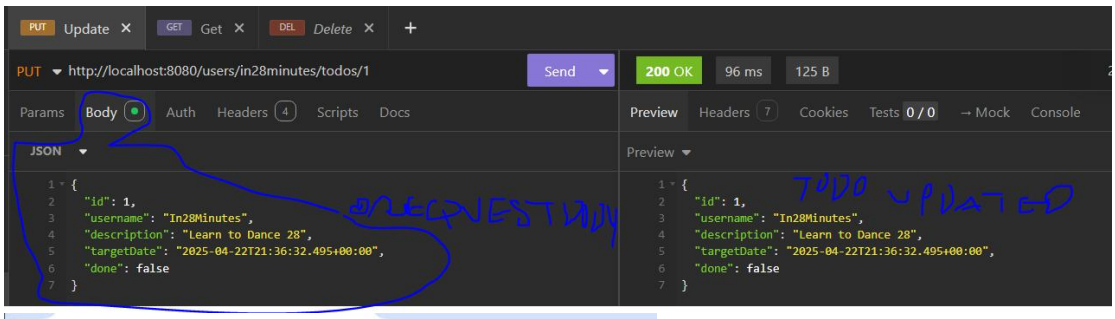


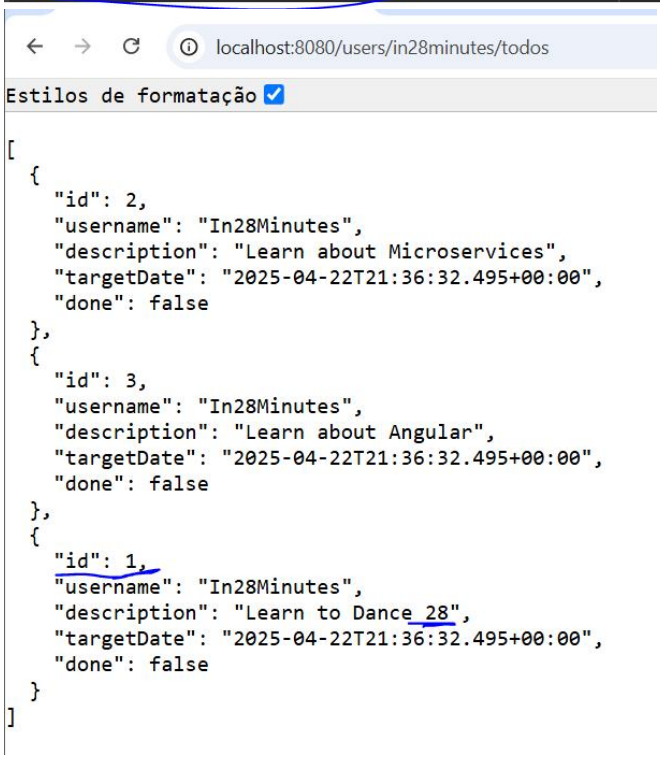
As in the database, the method save is used for insert / update some object, in the simulation with a static list for adding a new todo, it is verified if the id of the todo object is - 1 || 0, then replace the id by last position more one and actually insert it (.add(todo)). Otherwise, for updating the existing one, this one is deleted by its Id and a new todo is inserted with the same id but with different values.



Now, it is needed to map the request to update the todo(@PutMapping), receiving the variables through the path, as return include the Response Entity<Todo> object, add the todo object as a @RequestBody, use the service to make the call and inside the return make the instantiation passing the updated todo and the status of the response.







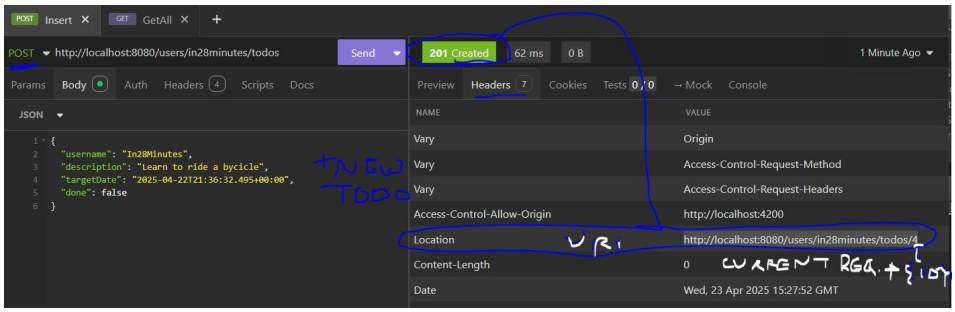
As a test without the front-end, it is made a get request to return the todo with id 1, copying this content, and on the put request paste the content as **the body of the request** for really updating the list and finally checking the whole list to see that the object is with the same id but with a different description.

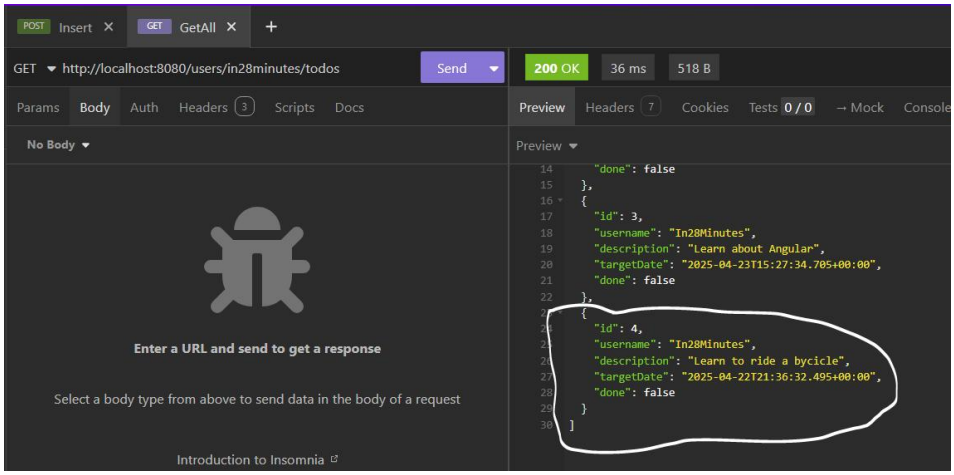
```
app.config.ts | todo.component.html | todo-data.service.ts | todo.component.ts
src > app > service > data > todo-data.service.ts > TodoDataService > updateTodo
  8  export class TodoDataService {
 25    updateTodo(username:string, id:number, todo:Todo){
 26      return this.http.put(`http://localhost:8080/users/${username}/todos/${id}`,
         todo);
 27    }
```

```
saveTodo(): void {
    this.todoService.updateTodo('in28minutes',this.id,this.todo)
    .subscribe(
      data => {
        console.log(data);
        this.router.navigate(['todos']);
      }
    );
}
```

Once the back-end is ready for updating, the front-end can make the call passing the variables and **the todo as a request body** on the put signature. When the user to click on the save button, this event will trigger the method saveTodo to make the call and navigate back to the todos component to see the todo already updated.

```
@PostMapping(path="/users/{username}/todos")
public ResponseEntity<Void> insertTodo(
        @PathVariable String username,
        @RequestBody Todo todo
        ){
    Todo insertedTodo = todoService.save(todo);

    URI uri = ServletUriComponentsBuilder.fromCurrentRequest()
            .path("/{id}").buildAndExpand(insertedTodo.getId()).toUri();

    return ResponseEntity.created(uri).build();
}
```

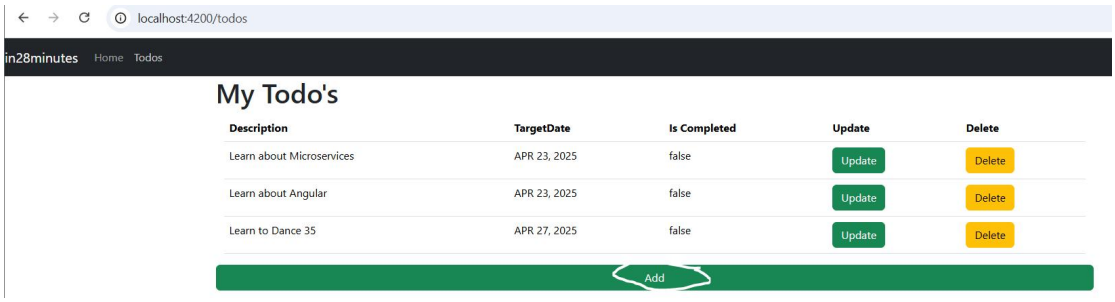




For inserting a new todo, it is mapped a new post request receiving the username as a variable and the todo object as a **@RequestBody**, then storing the inserted todo calling the save method from the service and return the status of created with the whole url including the new id. For building this uri object is used the class **ServletUriComponentsBuilder** to obtain the current path, include the {id} variable , replace it by the id from the todo object and convert the result as an URI value.

RESPONSE STATUS

- 200 - SUCCESS
- 404 - RESOURCE NOT FOUND
- 400 - BAD REQUEST
- 201 - CREATED
- 401 - UNAUTHORIZED
- 500 - SERVER ERROR



src > app > list-todos > TS list-todos.component.ts > ListTodosComponent

```
26  export class ListTodosComponent implements OnInit {
60
61    addTodo(){
62      this.router.navigate(['todos', -1]);
63    }
64  }
```

todo > TS todo.component.ts > TodoComponent > saveTodo > subscribe() callback

```
export class TodoComponent implements OnInit {
  ngOnInit(): void {
    this.id = this.route.snapshot.params['id'];
    if (this.id != -1) {
      this.todoService.retrieveTodo('in28minutes', this.id)
        .subscribe(data => this.todo = data);
    }
  }
```

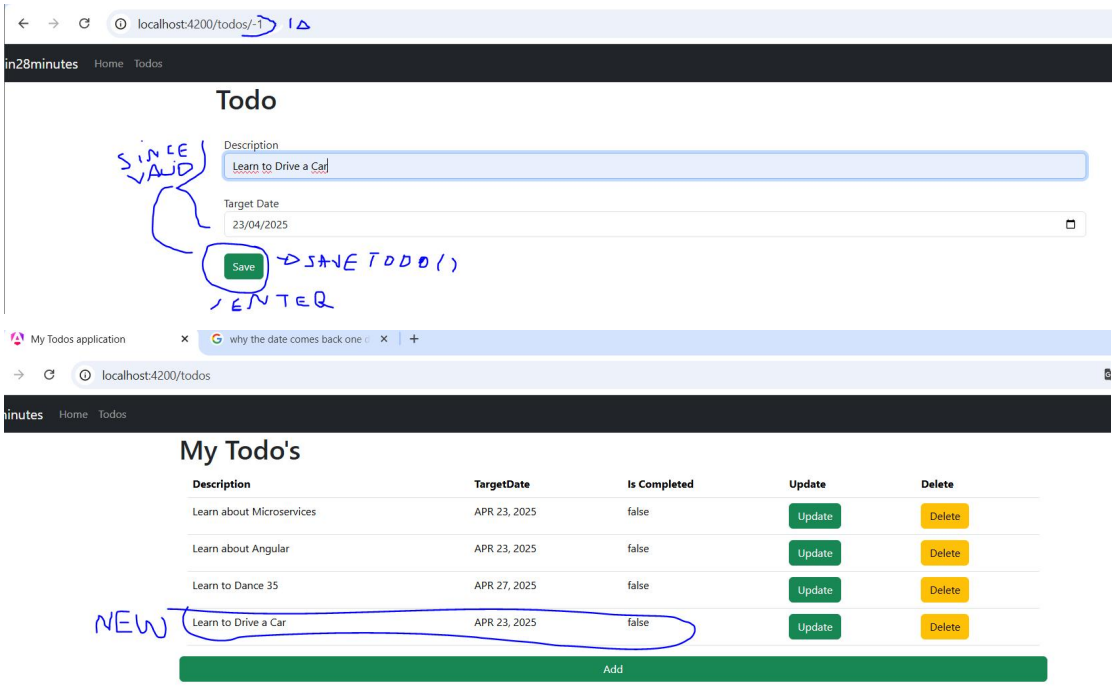
todo > TS todo.component.ts > TodoComponent > saveTodo > subscribe() callback

```
export class TodoComponent implements OnInit {
  saveTodo(): void {
    if (this.id == -1) {
      this.todoService.createTodo('in28minutes', this.todo)
      .subscribe(
        data => {
          console.log(data);
          this.router.navigate(['todos'])
        }
      )
    } else {
      this.todoService.updateTodo('in28minutes', this.id, this.todo)
        .subscribe(
          data => {
            console.log(data);
            this.router.navigate(['todos']);
```

service > data > TS todo-data.service.ts > TodoDataService > createTodo

```
export class TodoDataService {
  createTodo(username: string, todo: Todo) {
    return this.http.post(`http://localhost:8080/users/${username}/todos`, todo);
  }
}
```

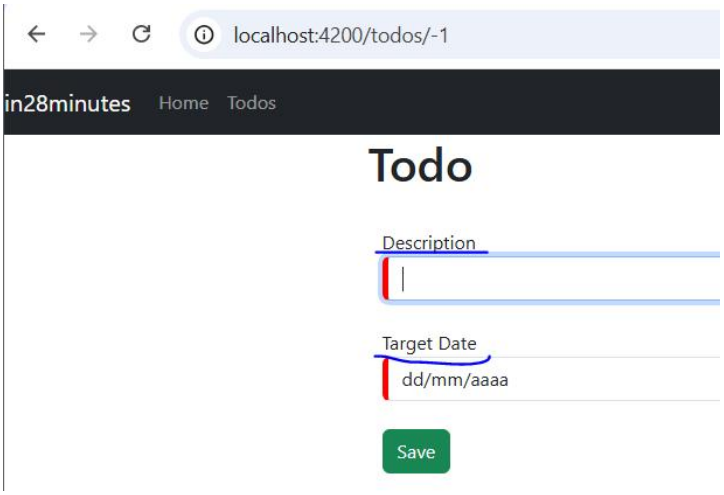
```
src > app > todo > todo.component.html > div.container > form
 3 <div class="container">
 4     <form (ngSubmit)="!todoForm.invalid && saveTodo()" #todoForm="ngForm">
10       <fieldset class="form-group">
16       </fieldset>
17       <br/>
18       <button type="submit" class="btn btn-success">Save</button>
19     </form>
20 </div>
```



On the front-end, it is included the button add triggering the method addTodo to navigate to the todo component **sending the id as -1.** On the todo component, the id is retrieved by the reference router of ActivatedRoute and verified before retrieving any data, because it's a new todo, then on the saveTodo method is verified if the id is -1 to call the service for adding a new one else call the update service for after doing the operation come back to the todo list to retrieve the new or updated todo.

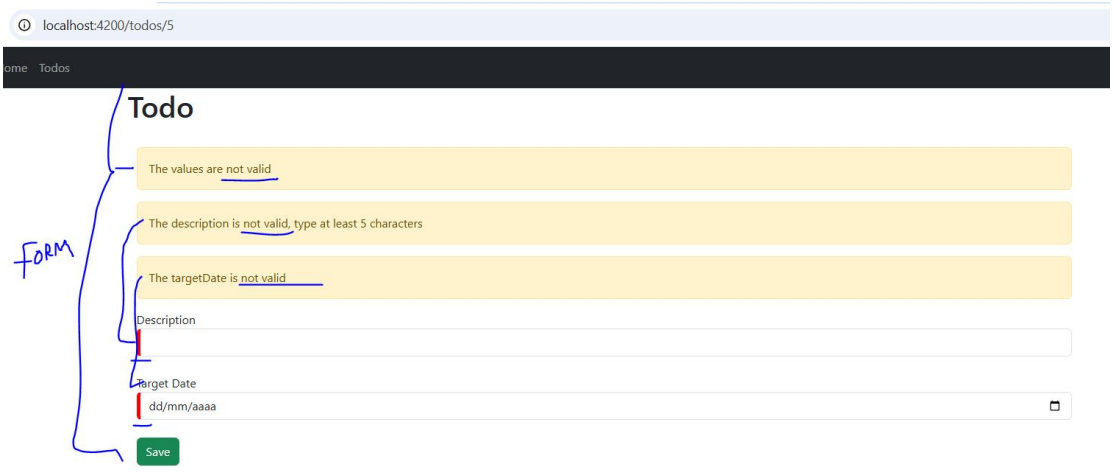
An important validation is to check if the fields are valid(not empty for instance) and be possible pressing the button Enter to send the form, for that is needed to include the tag form to make both fieldsets to be part it, the save button becomes a submit button (type), the tag form has an submit event **(ngSubmit)** since this form **is not invalid** for then triggering the method **saveTodo** using the id **#nameOfTheForm="ngForm"** to identify the form as an angular form.

```
src > app > todo > # todo.component.css > ...
1 .ng-invalid:not(form){
2   border-left: 5px solid red;
3 }
```




```
<div class="container">
  <div class="alert alert-warning" *ngIf="todoForm.dirty && todoForm.invalid">The values are
  not valid</div>
  <div class="alert alert-warning" *ngIf="description.dirty && description.invalid">The
  description is not valid, type at least 5 characters</div>
  <div class="alert alert-warning" *ngIf="targetDate.dirty && targetDate.invalid">The
  targetDate is not valid</div>
    <form (ngSubmit)="!todoForm.invalid && saveTodo()" #todoForm="ngForm">
      <fieldset class="form-group">
        <label for="description">Description</label>
        <input type="text" #description="ngModel" [(ngModel)]="todo.description"
        class="form-control" id="description" name="description" required="required"
        minlength="5"/>
      </fieldset>
      <br/>
```

```
    <fieldset class="form-group">
        <label for="targetDate">Target Date</label>
        <input type="date" #targetDate="ngModel"
        [ngModel]="todo.targetDate | date: 'yyyy-MM-dd'"
        (ngModelChange)="todo.targetDate = $event"
        class="form-control" id="targetDate" name="targetDate" required="required">
    </fieldset>
    <br/>
    <button type="submit" class="btn btn-success">Save</button>
  </form>
```

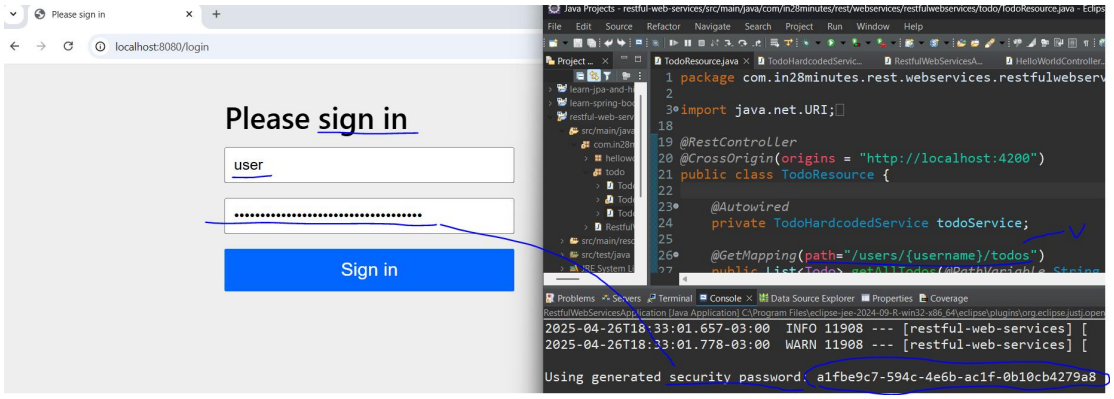


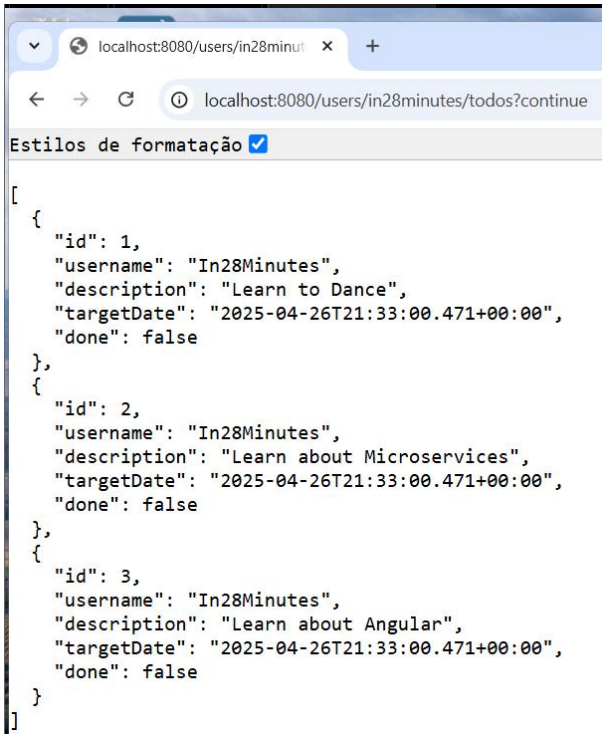
*dirty means is edited

Once the form is being validated it is important to display to the user why the form is not being sent, for that, every input needs to be **id**entified by a reference(#name=”directive”) and on a separated div use the *ngIf directive to check if is invalid “reference.invalid” >the message< is displayed. As a style, on the css, it is possible to individually catch each input by the class .ng-invalid:not(form){ adding a left border as solid and red with 5 pixels}.

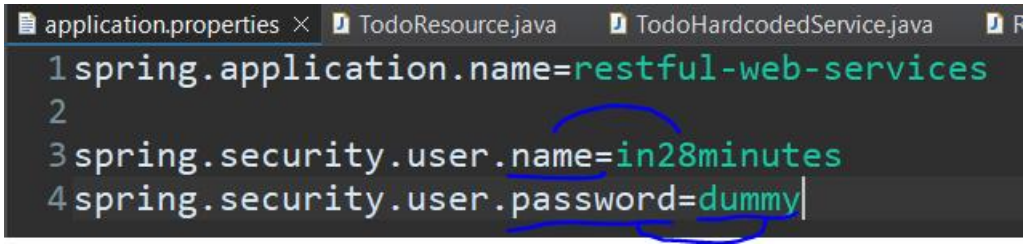
The section now is about security , because all the services exposed by the controller at the back-end is free without any authentication, thinking about it on SpringBoot there’s the dependency **spring security** and as feature to use the **J**ava **W**eb **T**oken that will serve as a session control besides username and password.

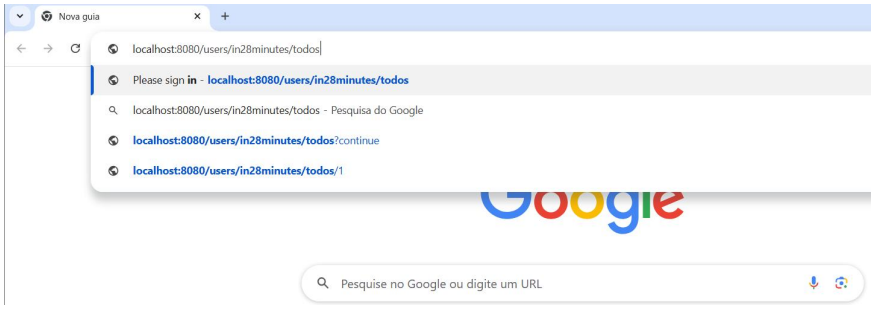
```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

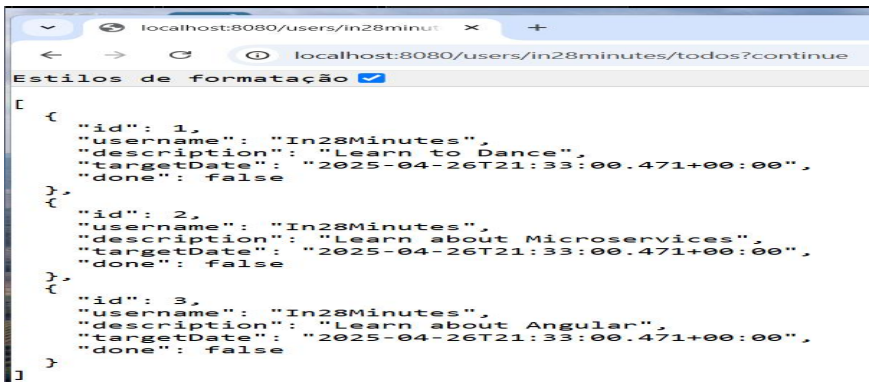




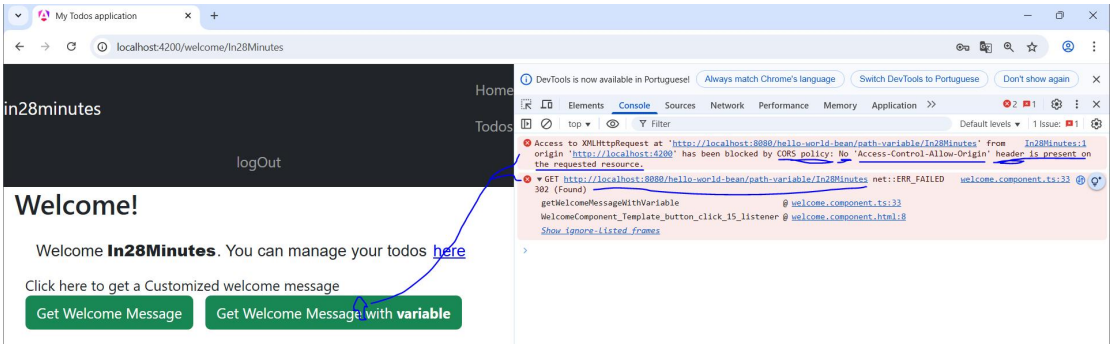
Just importing the spring security dependency (pom.xml), every first time accessing some service it is requested a **form based authentication with user and password** provided by spring boot and these are automatically sent encrypted in every request and connected to a cookie **session** until the browser closes.

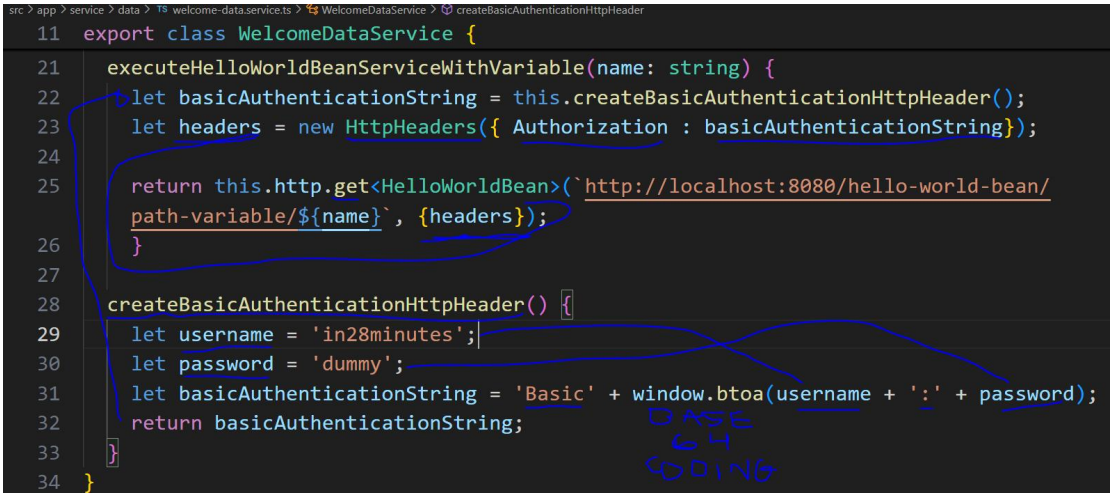




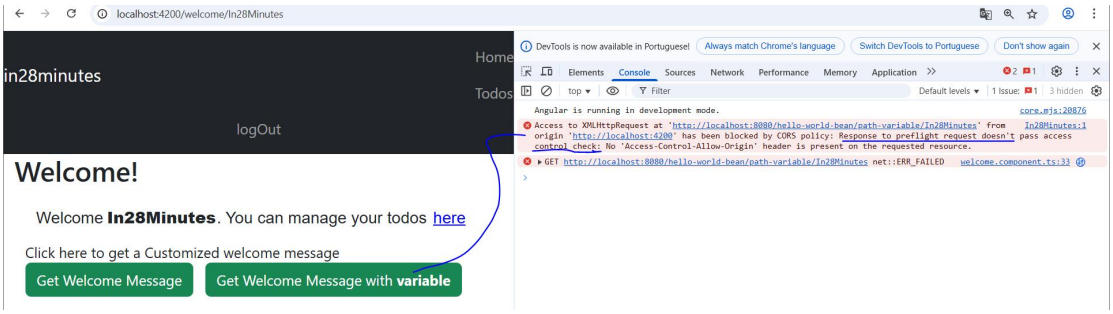


Editing the app.properties file it is possible to include a configuration to customize the username and password with the following ones: spring.security.user.name=”anything I want” and spring.security.user.password=”anything I want”.





*the ‘Basic’ without space is wrong replace by adding space ‘Basic ’



Now that the back-side is using the security dependency, there are some configurations that needed to be done, first send username and password in the request but encrypted joining the text ‘Basic ’ and the username and password coded in base 64 as parameter using the method window.btoa, then instantiate a new httpHeaders object receiving in the constructor the authorization property with the encrypted one to finally make the call using this httpHeaders object. With that the header is created, but still needs a few settings to allow the connection.

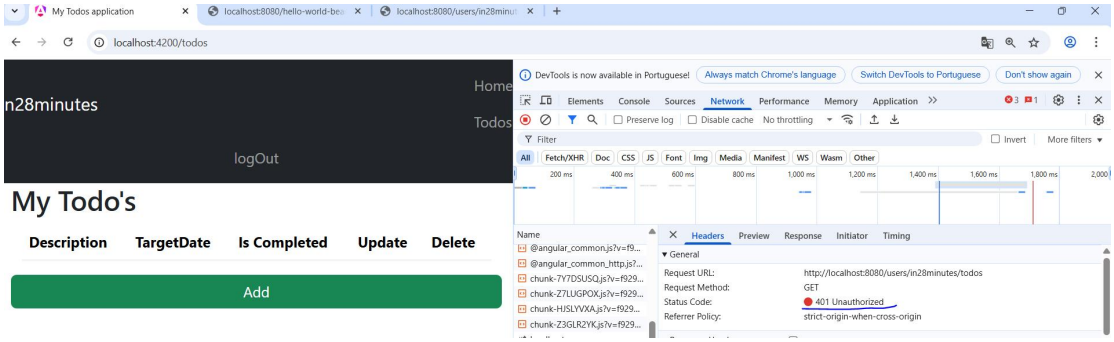
```java
@Configuration
@EnableWebSecurity
public class SpringSecurityConfigurationBasicAuth {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        return
                http
                    .authorizeHttpRequests(
                        auth ->
                            auth
                            .requestMatchers(HttpMethod.OPTIONS, "/**").permitAll()
                            .anyRequest().authenticated()
                    )
                    .httpBasic(Customizer.withDefaults())
                    .sessionManagement(
                        session -> session.sessionCreationPolicy
                        (SessionCreationPolicy.STATELESS))
                    .csrf((csrf) -> csrf.disable())
                    .build();
    }
}
```

As resolution for ‘ preflight request doesn't pass access control check’, is created a class responsible for setting the web security through a method that will return httpSecurity reference as an implementation of SecurityFilterChain interface to authorize the OPTIONS requests for any service to be attended and setting the basic authentication to disable the csrf protection without any session policy.

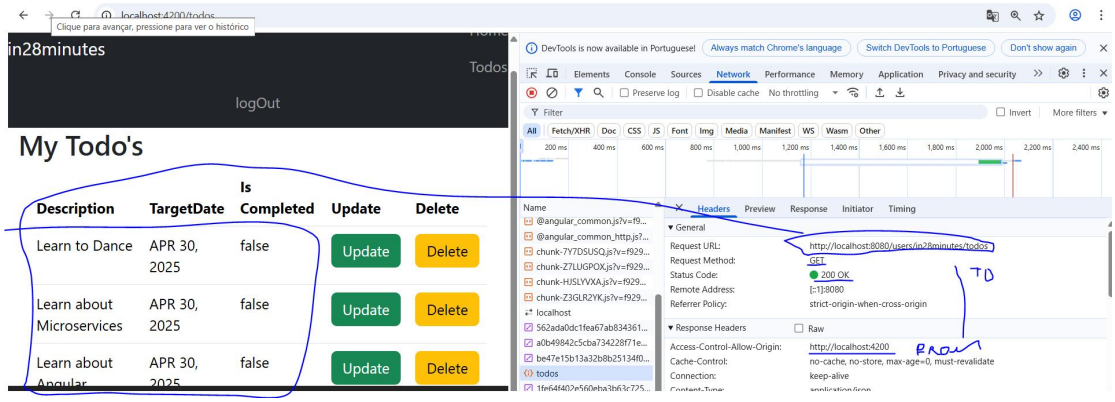
```
C:\Angular Projects\todo>ng generate service service/http/HttpInterceptorBasicAuth
CREATE src/app/service/http/http-interceptor-basic-auth.service.spec.ts (466 bytes)
CREATE src/app/service/http/http-interceptor-basic-auth.service.ts (162 bytes)
```

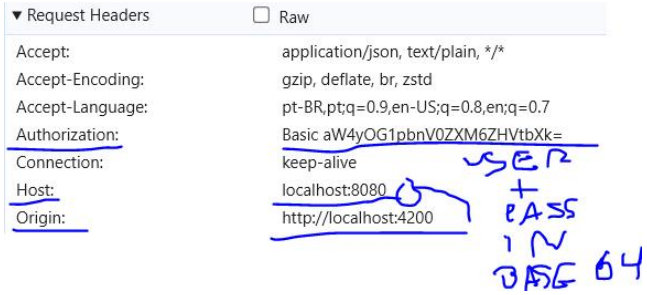
```typescript
export class HttpInterceptorBasicAuthService implements HttpInterceptor {
  constructor() { }

  intercept(request: HttpRequest<any>, next: HttpHandler) {
    let username = 'in28minutes';
    let password = 'dummy';
    let basicAuthenticationString = 'Basic ' + window.btoa(username + ':' + password);

    request = request.clone({
      setHeaders : {
        Authorization : basicAuthenticationString
      }
    })

    return next.handle(request);
  }
```

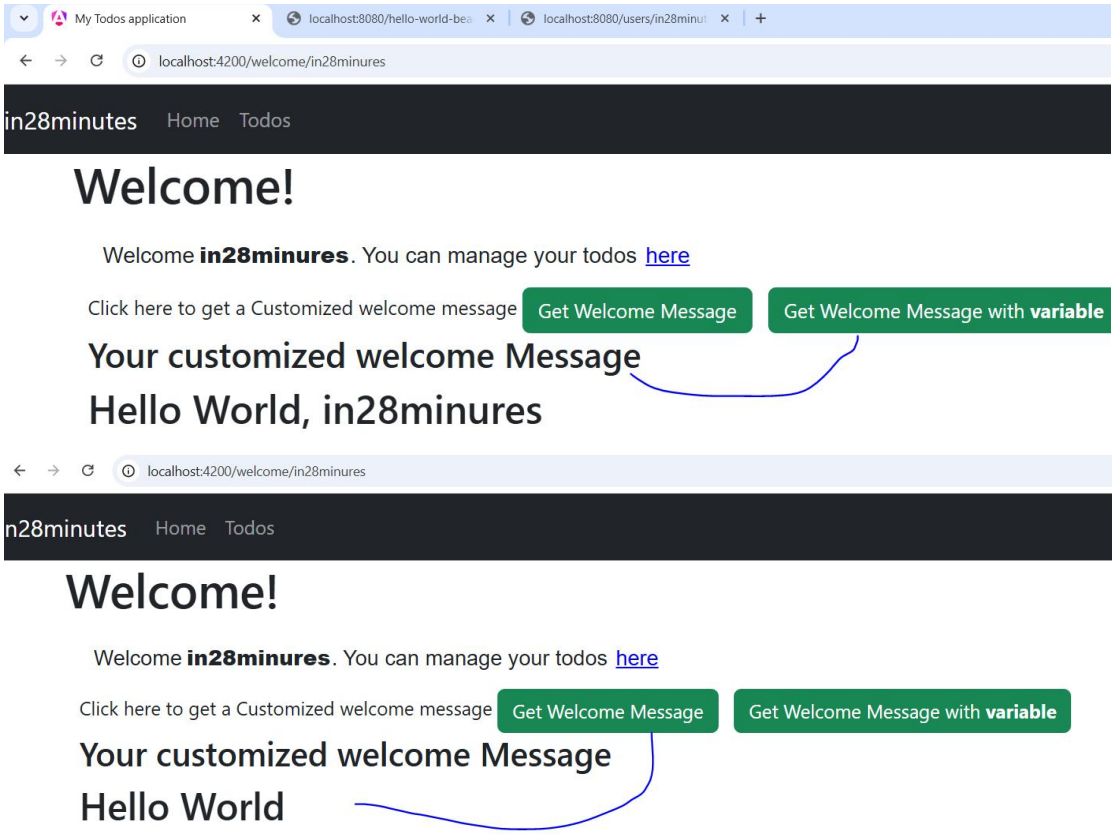
For not having to add manual header for each call, this logic can be reused and focused on a service called HttpInterceptorBasicAuth. This service will be responsible for implementing the HttpInterceptor interface and override the method intercept with the original parameters, pasting the username, password and the basicAuthenticationString properties. Then updating the request parameter with a clone of the original request and updating the headers with the property Authorization containing the basicAuthenticationString as value, the next parameter to handle **the new request**. In this way, any request will be intercepted and added with basic authentication.




```
src > app > app.config.ts > appConfig > providers
 7
 8  import {HTTP_INTERCEPTORS, provideHttpClient, withInterceptorsFromDi } from '@angular/common/
    http';
 9
10  import { HttpInterceptorBasicAuthService } from './service/http/http-interceptor-basic-auth.
    service';
11
12  export const appConfig: ApplicationConfig = {
13    providers: [
14      importProvidersFrom(BrowserModule,FormsModule),
15      provideRouter(routes),
16      {provide: HTTP_INTERCEPTORS, useClass: HttpInterceptorBasicAuthService, multi: true},
17      provideHttpClient(withInterceptorsFromDi())
18    ]
19  };
```







Now that it is configured, it is needed to include this interceptor on the app configuration in general way, **the field 'multi: true'** is important to add other interceptors in the future for not overriding the current one. Being possible to intercept every request and add the correct authorization with username and password, because without this basic authentication the request will be denied.

```
package com.in28minutes.rest.webservices.restfulwebservices.basic.auth;

import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@CrossOrigin(origins = "http://localhost:4200")
public class BasicAuthenticationController {

    @GetMapping(path = "/basicauth")
    public BasicAuthenticationBean basicAuth() {
        return new BasicAuthenticationBean("You are authenticated");
    }

}
```

For reformulating the hard-corded authentication on login through front-end, it is created one more controller accepting the front-end call and contemplating the mapping of a new get request called basicauth to return the message "you're authenticated".

```
src > app > service > basic-authentication.service.ts > BasicAuthenticationService > executeAuthenticationService > headers
 7  export class BasicAuthenticationService {
19    executeAuthenticationService(username:string, password: string){
20      let basicAuthHeaderString = 'Basic ' + window.btoa(username + ':'+password);
21      let headers = new HttpHeaders({
22        Authorization : basicAuthHeaderString
23      })
24      return this.http.get<AuthenticationBean>('http://localhost:8080/basicauth',
        {headers});
25    }
```

```
src > app > service > basic-authentication.service.ts > BasicAuthenticationService > getTokenAuthenticated
 8  export class BasicAuthenticationService {
48  }
49
50  export class AuthenticationBean {
51    constructor(public message: string) { }
52  }
53
```

The same service is created on the front-end receiving the username and password through parameter, coding them in base 64 joining with a ‘Basic ’, creating the HttpHeaders instance including the Authorization property with the coded one. Finally, returning the result of the get request having AuthenticationBean as a type and including the headers to it.

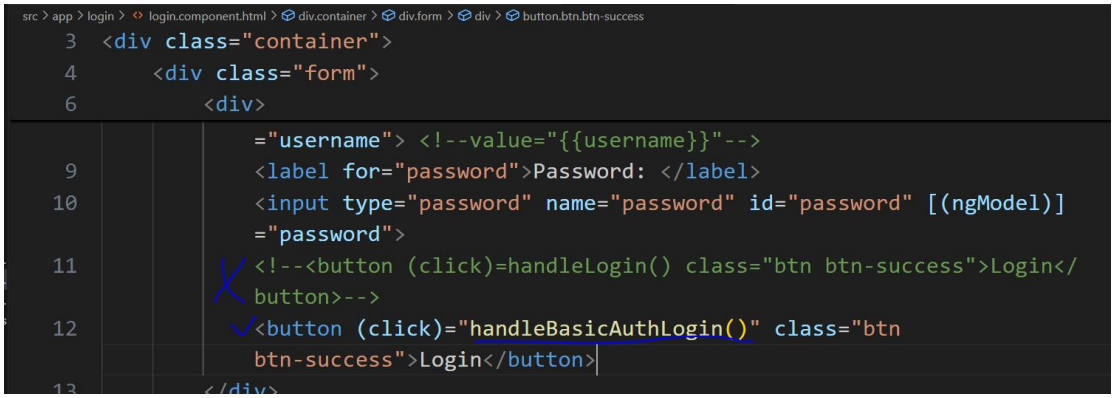
```
src > app > login > login.component.ts > LoginComponent > authenticationLogin > subscribe() callback
13  export class LoginComponent {
35    authenticationLogin(): void{
36      this.basicAuthenticationService.executeAuthenticationService(this.
        username, this.password).subscribe(
37        data => {
38          console.log(data);
39          this.router.navigate(['welcome', this.username]);
40          this.invalidLogin = false;
41        },
42        error =>{
43          console.log(error);
44          this.invalidLogin = true;
45        }
46      )
47    }
```

```
src > app > service > basic-authentication.service.ts > BasicAuthenticationService > executeAuthenticationService
 8  export class BasicAuthenticationService {
20    executeAuthenticationService(username: string, password: string) {
22      let headers = new HttpHeaders({
23        Authorization: basicAuthHeaderString
24      })
25      return this.http.get<AuthenticationBean>('http://localhost:8080/basicauth',
        { headers }).pipe(
26        map(
27          data => {
28            sessionStorage.setItem("UserAuthenticated", username);
29            return data;
30          }
31        )
32      );
33    }
```

On the login component (.ts), it is included the method authenticationLogin to call the basicAuthService method for making the request in fact passing the username and password

as parameters, subscribing it,in case of success retrieve the data, redirect the user to the welcome page and make the login valid the other way around retrieve the error data and invalid the login.

On the service, at the end of the get call method, **improve the observable object(plus)** with the method **pipe**, map the success data, include the username on the sessionStorage as user authenticated.



On the login template, create one more button to call this new authentication to successfully make the login or not, because the username and password sent will be compared with the ones on the properties file in the back-end side when the request is done. Just a point, the http interceptor was commented out(appConfig.ts).

As a refactoring of basicAuthenticationService, include the basicAuthHeaderString as a token in the session, also create the get methods for both items, but to recover the token the user needs to be authenticated.

Having the token and the username saved in the session storage and recoverable by it, the interceptor needs to recover them and save on variables to check if the login have already been done, if true, capture the original request then update the headers to include an object with the authorization attribute with the token else the request goes on without it. This way, the requests in general won't have to worry about authentication.

src > app > service > data > todo-data.service.ts > TodoDataService > retrieveTodo

```
9  export class TodoDataService {
14   retrieveAllTodos(username: string) {
15     return this.http.get<Todo[]>(`${API_URL}/users/${username}/todos`);
16   }
17
18   deleteTodo(username: string, id: number) {
19     return this.http.delete(`${API_URL}/users/${username}/todos/${id}`);
20   }
21
22   retrieveTodo(username: string, id: number) {
23     return this.http.get<Todo>(`${API_URL}/users/${username}/todos/${id}`);
24   }
25
26   updateTodo(username: string, id: number, todo: Todo) {
27     return this.http.put(`${API_URL}/users/${username}/todos/${id}`, todo);
28   }
```

```
export const TOKEN = 'token';
export const USER_AUTHENTICATED = 'UserAuthenticated';

@Injectable({
  providedIn: 'root'
})
export class BasicAuthenticationService {
```

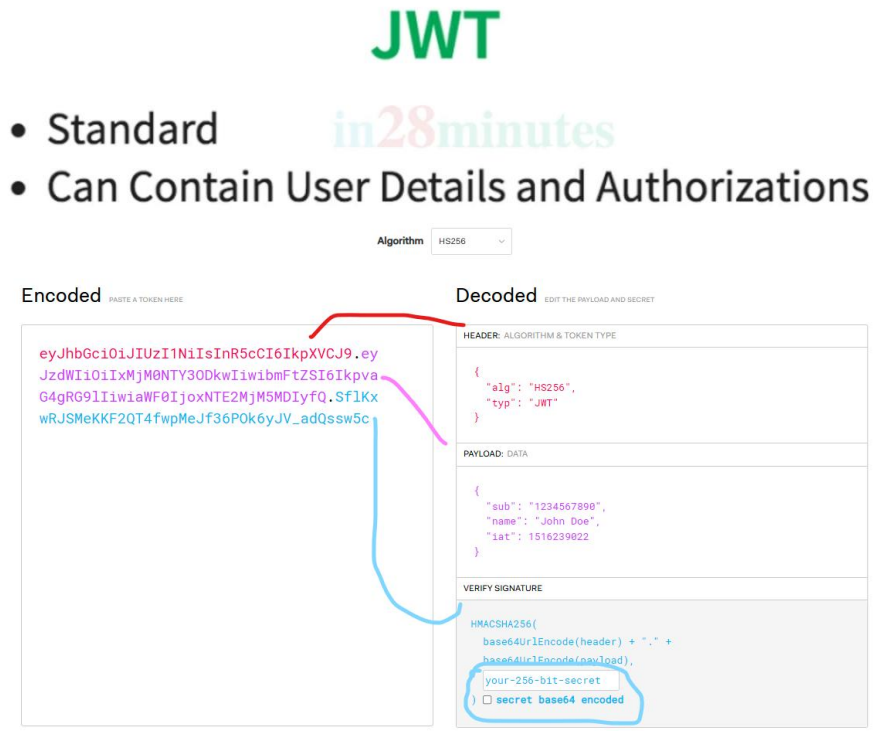
```
  return this.http.get<AuthenticationBean>(`${API_URL}/basicauth`, { headers }).pipe(
    map(
      data => {
        sessionStorage.setItem(USER_AUTHENTICATED, username);
        sessionStorage.setItem(TOKEN, basicAuthHeaderString);
        return data;
      }
    )
  );
}
```

A good practice is to use constants for values that are **frequently used and don't change**, avoiding misspelling when editing different files.

BASIC AUTHENTICATION

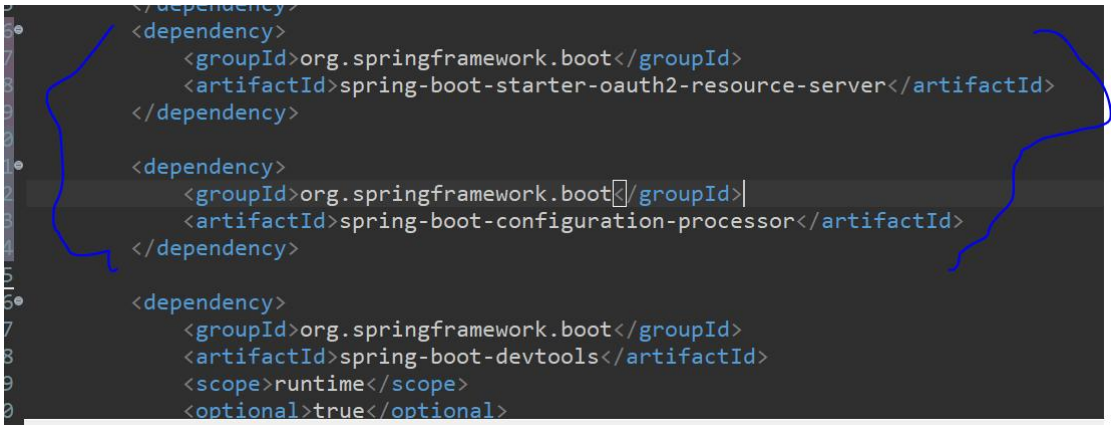
- No Expiration Time
- No User Details

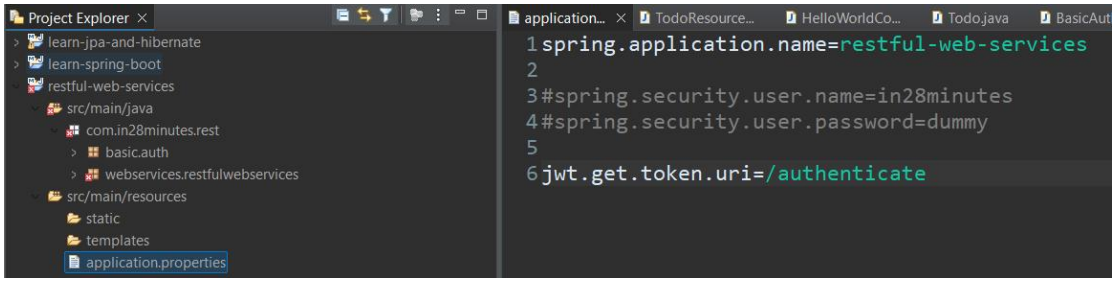
ONLY USER + PASS



JWT (**J**ava **W**eb **T**oken) is a **security standard** with expiration time like a clock session that brings up one more security measure on the **user authentication** having user data, authorization data using an algorithm to encode the information in a non-understandable sequence of numbers and letters. As a key to protect the data, the field ‘your-256-bit-secret’ is open to insert a **customized riddle** that only the **authorized** systems & people must know to get access to the token.

*IAT-> token creation time





For Implementing the JWT, the dependencies oauth-2-resource-server and configuration-processor are needed on pom.xml. Secondly, on application.properties include the new route for authenticating the user using **jwt**.get.token.uri=/authenticate. As part of the classes structure to create the unique token to be used in every request, it is needed to include the following ones:

```
package com.in28minutes.rest.webservices.restfulwebservices.jwt;

public record JwtTokenRequest(String username, String password) {}
```

```
package com.in28minutes.rest.webservices.restfulwebservices.jwt;

public record JwtTokenResponse(String token) {}
```

```
package com.in28minutes.rest.webservices.restfulwebservices.jwt;

import java.time.Instant;
import java.time.temporal.ChronoUnit;
import java.util.stream.Collectors;

import org.springframework.security.core.Authentication;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.oauth2.jwt.JwtClaimsSet;
import org.springframework.security.oauth2.jwt.JwtEncoder;
import org.springframework.security.oauth2.jwt.JwtEncoderParameters;
import org.springframework.stereotype.Service;

@Service
public class JwtTokenService {

    private final JwtEncoder jwtEncoder;

    public JwtTokenService(JwtEncoder jwtEncoder) {
```

```
        this.jwtEncoder = jwtEncoder;
    }

    public String generateToken(Authentication authentication) {

        var scope = authentication
                    .getAuthorities()
                    .stream()
                    .map(GrantedAuthority::getAuthority)
                    .collect(Collectors.joining(" "));

        var claims = JwtClaimsSet.builder()
                    .issuer("self")
                    .issuedAt(Instant.now())
                    .expiresAt(Instant.now().plus(90, ChronoUnit.MINUTES))
                    .subject(authentication.getName())
                    .claim("scope", scope)
                    .build();

        return this.jwtEncoder
                .encode(JwtEncoderParameters.from(claims))
                .getTokenValue();
    }
}
```

```
package com.in28minutes.rest.webservices.restfulwebservices.jwt;

import org.springframework.http.ResponseEntity;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

@RestController
@CrossOrigin(origins="http://localhost:4200")
public class JwtAuthenticationController {
```

```
    private final JwtTokenService tokenService;

    private final AuthenticationManager authenticationManager;

    public JwtAuthenticationController(JwtTokenService tokenService,
            AuthenticationManager authenticationManager) {
        this.tokenService = tokenService;
        this.authenticationManager = authenticationManager;
    }

    @PostMapping("/authenticate")
    public ResponseEntity<JwtTokenResponse> generateToken(
            @RequestBody JwtTokenRequest jwtTokenRequest) {

        var authenticationToken =
                new UsernamePasswordAuthenticationToken(
                        jwtTokenRequest.username(),
                        jwtTokenRequest.password());

        var authentication =
                authenticationManager.authenticate(authenticationToken);

        var token = tokenService.generateToken(authentication);

        return ResponseEntity.ok(new JwtTokenResponse(token));
    }
}
```

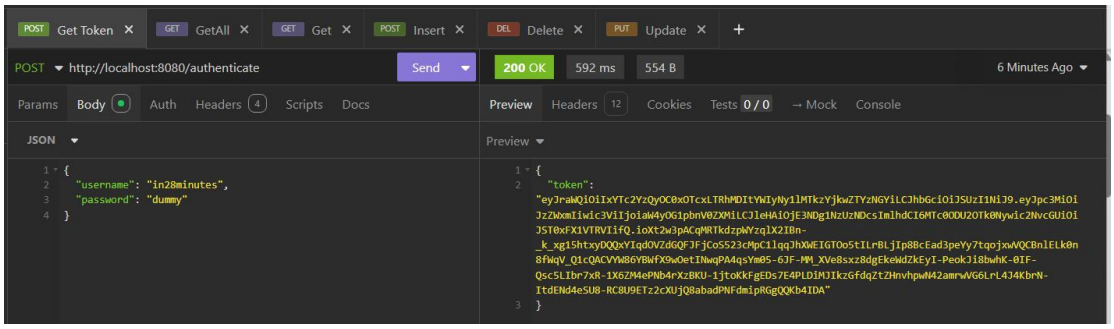
```
package com.in28minutes.rest.webservices.restfulwebservices.jwt;

import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.interfaces.RSAPrivateKey;
import java.security.interfaces.RSAPublicKey;
import java.util.UUID;

import org.springframework.boot.autoconfigure.security.servlet.PathRequest;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
```

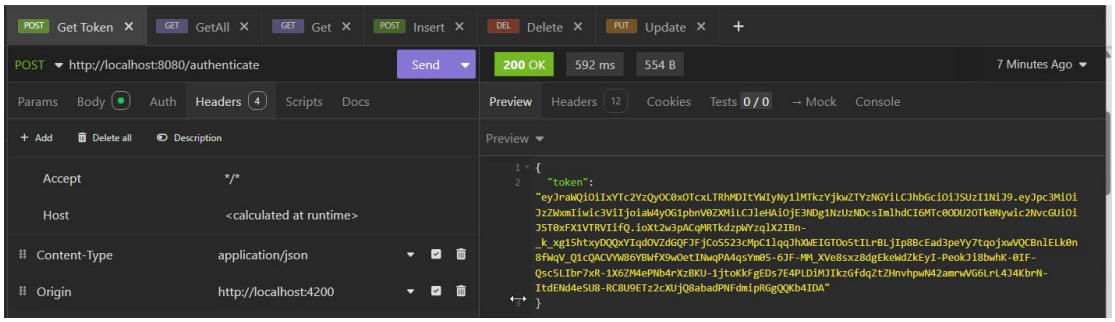
```
import org.springframework.http.HttpMethod;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.ProviderManager;
import org.springframework.security.authentication.dao.DaoAuthenticationProvider;
import org.springframework.security.config.Customizer;
import
org.springframework.security.config.annotation.method.configuration.EnableMethodSecurity;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import
org.springframework.security.config.annotation.web.configurers.AbstractHttpConfigurer;
import
org.springframework.security.config.annotation.web.configurers.oauth2.server.resource.OAut
h2ResourceServerConfigurer;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.oauth2.jwt.JwtDecoder;
import org.springframework.security.oauth2.jwt.JwtEncoder;
import org.springframework.security.oauth2.jwt.NimbusJwtDecoder;
import org.springframework.security.oauth2.jwt.NimbusJwtEncoder;
import org.springframework.security.provisioning.InMemoryUserDetailsManager;
import org.springframework.security.web.SecurityFilterChain;
import org.springframework.web.servlet.handler.HandlerMappingIntrospector;

import com.nimbusds.jose.JOSEException;
import com.nimbusds.jose.jwk.JWKSet;
import com.nimbusds.jose.jwk.RSAKey;
import com.nimbusds.jose.jwk.source.JWKSource;
import com.nimbusds.jose.proc.SecurityContext;

import static org.springframework.security.config.Customizer.withDefaults;
@Configuration
@EnableWebSecurity
@EnableMethodSecurity
public class JwtSecurityConfig {

    @Bean
```

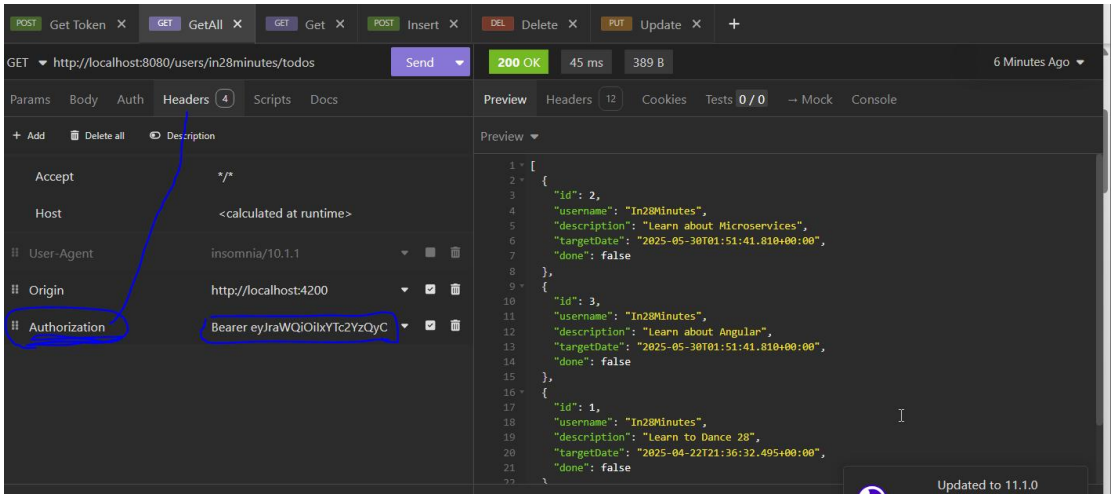
```
  public SecurityFilterChain securityFilterChain(HttpSecurity httpSecurity,
HandlerMappingIntrospector introspector) throws Exception {

    // h2-console is a servlet
    // https://github.com/spring-projects/spring-security/issues/12310
    return httpSecurity
        .authorizeHttpRequests(auth -> auth
          .requestMatchers("/authenticate").permitAll()
          .requestMatchers(PathRequest.toH2Console()).permitAll() // h2-console is a
servlet and NOT recommended for a production
          .requestMatchers(HttpMethod.OPTIONS,"/**")
          .permitAll()
          .anyRequest()
          .authenticated())
        .csrf(AbstractHttpConfigurer::disable)
        .sessionManagement(session -> session.
          sessionCreationPolicy(SessionCreationPolicy.STATELESS))
        .oauth2ResourceServer(
            (oauth2) -> oauth2.jwt(withDefaults()))
        .httpBasic(
            Customizer.withDefaults())
        .headers(headers -> headers.frameOptions(frameOptionsConfig->
frameOptionsConfig.disable()))
        .build();
  }

  @Bean
  public AuthenticationManager authenticationManager(
      UserDetailsService userDetailsService) {
    var authenticationProvider = new DaoAuthenticationProvider();
    authenticationProvider.setUserDetailsService(userDetailsService);
    return new ProviderManager(authenticationProvider);
  }

  @Bean
  public UserDetailsService userDetailsService() {
    UserDetails user = User.withUsername("in28minutes")
                .password("{noop}dummy")
                .authorities("read")
                .roles("USER")
```

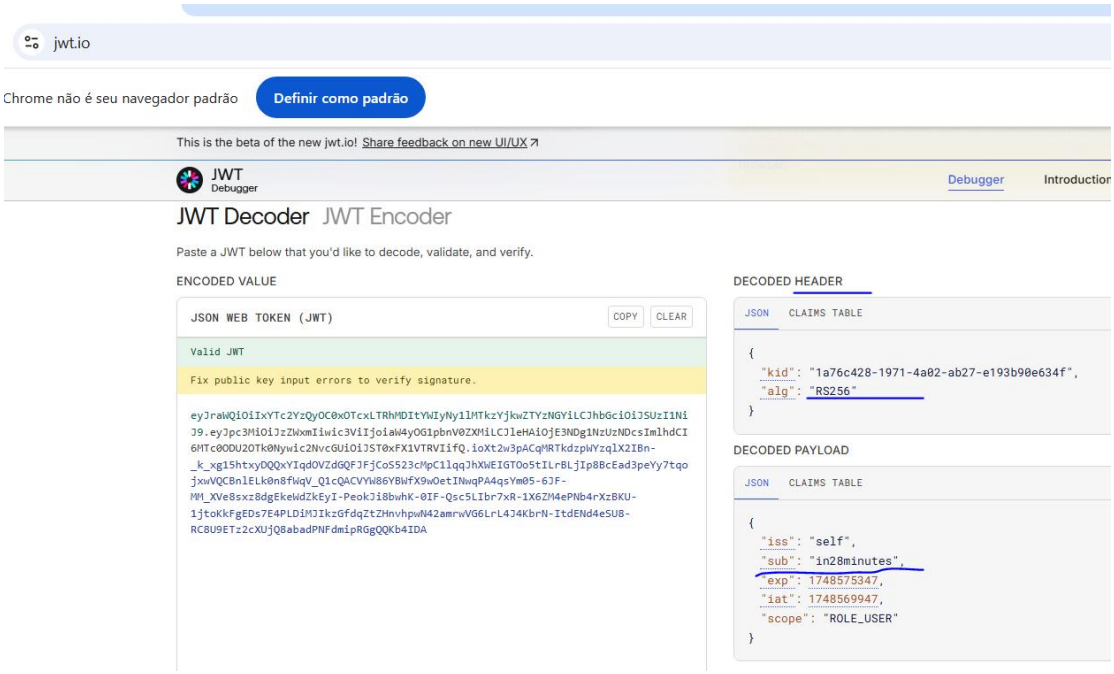
```
                .build();

    return new InMemoryUserDetailsManager(user);
}

@Bean
public JWKSource<SecurityContext> jwkSource() {
    JWKSet jwkSet = new JWKSet(rsaKey());
    return (((jwkSelector, securityContext)
                -> jwkSelector.select(jwkSet)));
}

@Bean
JwtEncoder jwtEncoder(JWKSource<SecurityContext> jwkSource) {
    return new NimbusJwtEncoder(jwkSource);
}

@Bean
JwtDecoder jwtDecoder() throws JOSEException {
    return NimbusJwtDecoder
            .withPublicKey(rsaKey().toRSAPublicKey())
            .build();
}

@Bean
public RSAKey rsaKey() {

    KeyPair keyPair = keyPair();

    return new RSAKey
            .Builder((RSAPublicKey) keyPair.getPublic())
            .privateKey((RSAPrivateKey) keyPair.getPrivate())
            .keyID(UUID.randomUUID().toString())
            .build();
}

@Bean
public KeyPair keyPair() {
    try {
        var keyPairGenerator = KeyPairGenerator.getInstance("RSA");
```

```
            keyPairGenerator.initialize(2048);
            return keyPairGenerator.generateKeyPair();
        } catch (Exception e) {
            throw new IllegalStateException(
                    "Unable to generate an RSA Key Pair", e);
        }
    }

}
```










```java
@Service
public class JwtTokenService {

    private final JwtEncoder jwtEncoder;

    public JwtTokenService(JwtEncoder jwtEncoder) {
        this.jwtEncoder = jwtEncoder;
    }
```

```java
var claims = JwtClaimsSet.builder()
                .issuer("self")
                .issuedAt(Instant.now())
                .expiresAt(Instant.now().plus(90, ChronoUnit.MINUTES))
                .subject(authentication.getName())
                .claim("scope", scope)
                .build();
```

Run it on insomnia on post /authenticate route to obtain the token then make a request to obtain the list of todos of the user using the field Authorization with the **obtained token** as Bearer token value. Checking it on jwt.io, the token is validated.

```
export class BasicAuthenticationService {
  constructor(private http: HttpClient) { }

  executeJWTAuthenticationService(username:string, password:string ){

    return this.http.post<any>(`${API_URL}/authenticate`,{
      username,
      password
    }).pipe(
      map (
        data => {
          sessionStorage.setItem(USER_AUTHENTICATED, username);
          sessionStorage.setItem(TOKEN, `Bearer ${data.token}`);
          return data;
        }
      )
```

```
export class LoginComponent {
  // dependency injection (@Autowired)
  constructor(private router: Router, private basicAuthenticationService :
  BasicAuthenticationService){}

  handleJWTAuthLogin(): void{
      this.basicAuthenticationService.executeJWTAuthenticationService(this.username, this.
      password).subscribe(
        data => {
          console.log(data);
          this.router.navigate(['welcome', this.username]);
          this.invalidLogin = false;
```

```
  <label for="password">Password: </label>
  <input type="password" name="password" id="password" [(ngModel)]="password">
<button (click)="handleJWTAuthLogin()" class="btn btn-success"> Login</button>
```

Applying this call to the front-end, it is needed in a new method to make a post request to /authenticate route, passing the requestBody with username and password, retrieve the successful response data, use the data to store the token on the sessionStorage as an item as ‘Bearer ${data.token} and the username. Then, update the method called by the login button to use the JWT authentication and the button itself to call the updated method. This way, every time the interceptor class include the authorization, it will contain the updated token.

```java
package com.in28minutes.rest.webs

import java.util.Date;

@Entity
public class Todo {

    @Id
    @GeneratedValue
    private Long id;
    private String username;
    private String description;
    private Date targetDate;
    private boolean isDone;

```

Applying the JPA mapping for defining the table of the database, it is needed to use the @annotations over the table class to identify the class and its fields.
Description:
@Entity-> Relational Object Mapping (class to table)
@Table(name=”customized table name”)
@Id-> primary key
@GeneratedValue-> auto increment from 1 and plus 1
@Column(name=”customized field name”)

data.sql

C: > Java Projects > restful-web-services > restful-web-services > src > main > resources > data.sql

```sql
insert into todo(id,username,description,target_date,is_done)
values(1001,'in28minutes','Learn SpringBoot',CURRENT_TIMESTAMP,false);

insert into todo(id,username,description,target_date,is_done)
values(1002,'in28minutes','Learn Spring JPA',CURRENT_TIMESTAMP,false);

insert into todo(id,username, description, target_date,is_done)
values(1003,'in28minutes','Learn Microservices',CURRENT_TIMESTAMP,false);
```

On the data.sql inside resources, it is possible to pre-populate the data todos using the SQL commands.

```
spring.datasource.url=jdbc:h2:mem:testdb;NON_KEYWORDS=USER
spring.h2.console.path=/h2-console
spring.h2.console.enabled=true
spring.jpa.show-sql=true
spring.jpa.defer-datasource-initialization=true
spring.data.jpa.repositories.bootstrap-mode=default
```

Also important, define the url of the database, enable the h2 on the browser and its uri, show the sql used to see the database and tables creation in terminal.

Once the table is mapped, it is needed to make a new controller to service these calls to the database then the TodoJpaResource is created as a copy of the TodoResource class, just putting '/jpa' at the beginning of each request mapping uri.

```java
package com.in28minutes.rest.webservices.restfulwebservices.todo;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface TodoJpaRepository extends JpaRepository<Todo,Long> {

    List<Todo> findByUsername(String username);
}
```

The interfaceTodoJpaRepository is created to manipulate the operations related to the Todo table, adding the annotation @Repository to identify it as a data access layer. Then, it extends from JpaRepository passing the entity Todo and the Id type as generic type, also create the method findByUsername receiving the username as parameter and returning a List of todos as answer.

Still on TodoJpaResource, create the instance of the interface with the annotation @Autowired, inside of getTodo method use this instance to return one Todo by Id and how the return is an Optional, it is needed to use the method .get() to obtain the original object (Todo) and inside of getAllTodos method use this interface to return the list of todos according to the username.

```
src > app > app.constants.ts > TODO_JPA_API_URL
1  export const API_URL = 'http://localhost:8080';
2  export const TODO_JPA_API_URL = 'http://localhost:8080/jpa'
```

```
src > app > service > data > todo-data.service.ts > TodoDataService > retrieveTodo
9  export class TodoDataService {
12     private http: HttpClient) { }
13
14   retrieveAllTodos(username: string) {
15     return this.http.get<Todo[]>(`${TODO_JPA_API_URL}/users/${username}/todos`);
16   }
17
18   deleteTodo(username: string, id: number) {
19     return this.http.delete(`${API_URL}/users/${username}/todos/${id}`);
20   }
21
22   retrieveTodo(username: string, id: number) {
23     return this.http.get<Todo>(`${TODO_JPA_API_URL}/users/${username}/todos/${id}`);
```

On the front-end, update the TodoDataService class using a new constant to include the '/jpa' at the beginning of the http call.

```
application.properties | data.sql | TodoJpaRepository.java | TodoJpaResource.java | TodoHardcodedService.java
34     }
35
36     @DeleteMapping(path="/jpa/users/{username}/todos/{id}")
37     public ResponseEntity<Void> deleteTodo(
38             @PathVariable String username,
39             @PathVariable long id){
40
41         todoJpaRepository.deleteById(id);
42         return ResponseEntity.noContent().build();
43         //return ResponseEntity.notFound().build();
44     }
45
46     @PutMapping(path="/jpa/users/{username}/todos/{id}")
47     public ResponseEntity<Todo> updateTodo(
48             @PathVariable String username,
49             @PathVariable long id,
50             @RequestBody Todo todo){
51         Todo todoUpdated = todoJpaRepository.save(todo);
52
53         return new ResponseEntity<Todo>(todoUpdated, HttpStatus.OK);
54     }
55
```

```
@PostMapping(path="/jpa/users/{username}/todos")
public ResponseEntity<Void> insertTodo(
        @PathVariable String username,
        @RequestBody Todo todo
        ){

    todo.setUsername(username);
    todo.setId(null);
    Todo insertedTodo = todoJpaRepository.save(todo);

    URI uri = ServletUriComponentsBuilder.fromCurrentRequest()
            .path("/{id}").buildAndExpand(insertedTodo.getId()).toUri();

    return ResponseEntity.created(uri).build();
}
```

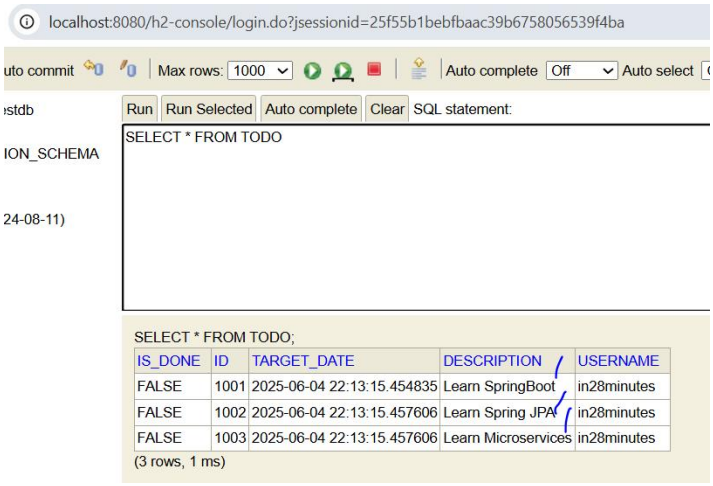
For finishing the application as a whole, it is needed to update the rest of the request mappings on back-end side, replacing the call from todoService to todoJpaRepository. Just two observations: the not found content was commented because if the one is not found, an exception is thrown, the username had to be overridden for not being null at database and the id had

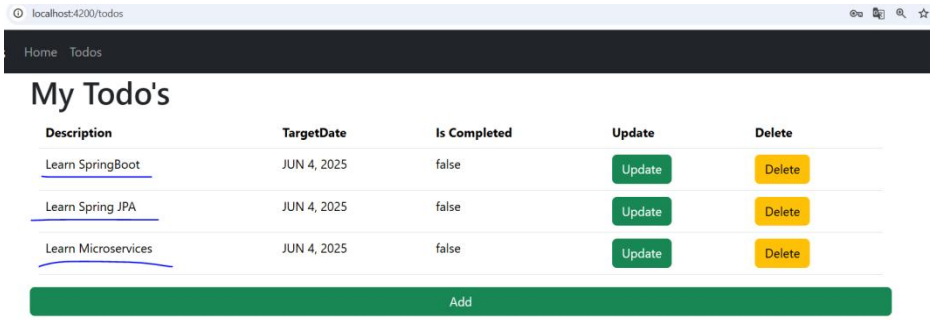
to be overridden because it can't be 0 or -1 given that this field is auto incremented, so it has to be null before inserting in fact by the interface for that hibernate can fulfil it as the last inserted plus 1.

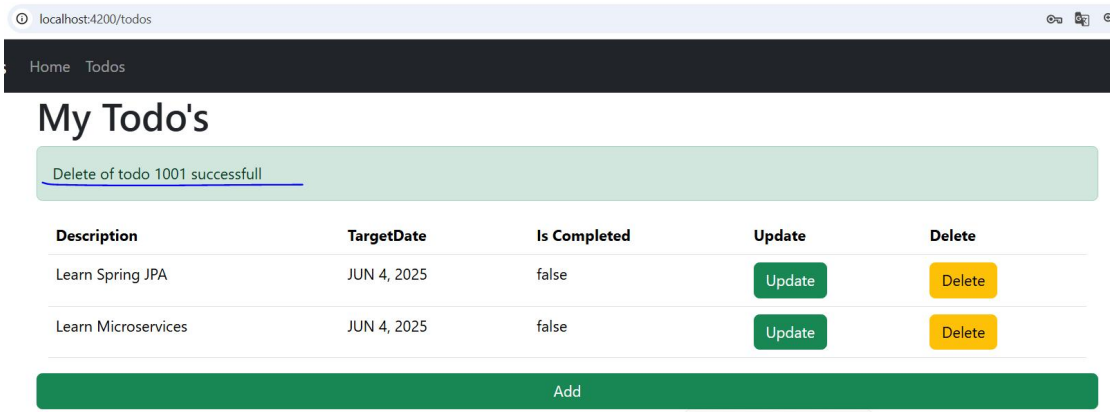
```
retrieveAllTodos(username: string) {
  return this.http.get<Todo[]>(`${TODO_JPA_API_URL}/users/${username}/todos`);
}
```

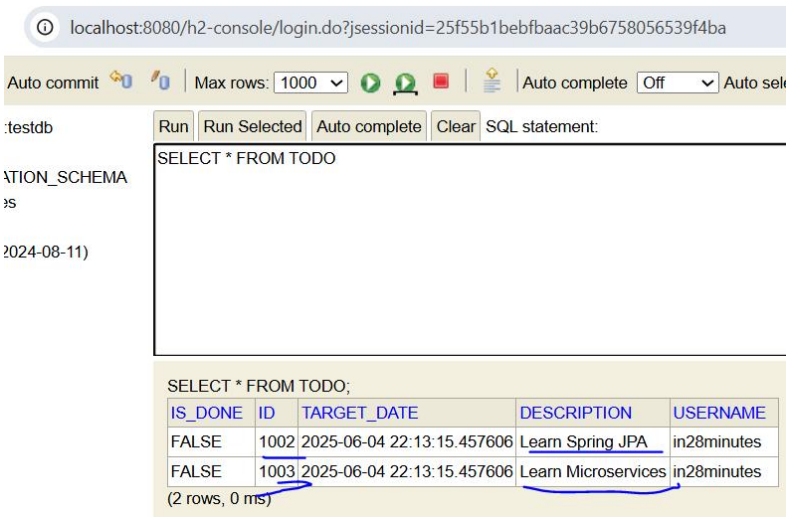
```
src > app > service > data > todo-data.service.ts > TodoDataService > createTodo
 9  export class TodoDataService {
18    deleteTodo(username: string, id: number) {
19      return this.http.delete(`${TODO_JPA_API_URL}/users/${username}/todos/${id}`);
20    }
21
22    retrieveTodo(username: string, id: number) {
23      return this.http.get<Todo>(`${TODO_JPA_API_URL}/users/${username}/todos/${id}`);
24    }
25
26    updateTodo(username: string, id: number, todo: Todo) {
27      return this.http.put(`${TODO_JPA_API_URL}/users/${username}/todos/${id}`, todo);
28    }
29
30    createTodo(username: string, todo: Todo) {
31      return this.http.post(`${TODO_JPA_API_URL}/users/${username}/todos`, todo);
32    }
```

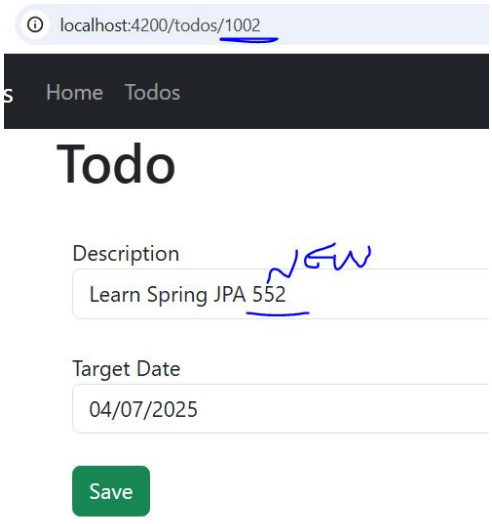
The same on the front-end, putting all the constants considering the '/jpa'. At this point the application is done.

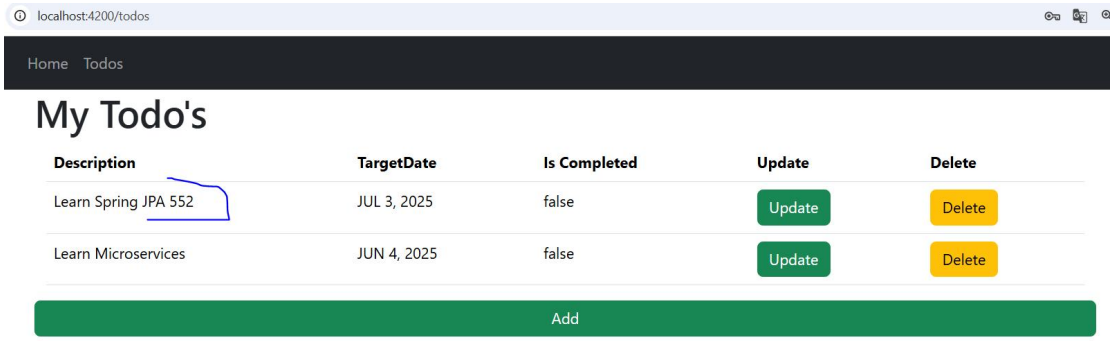


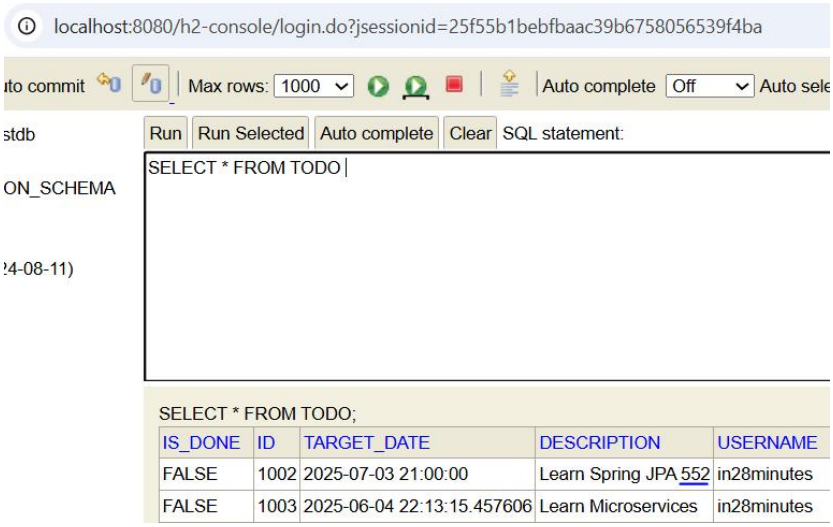


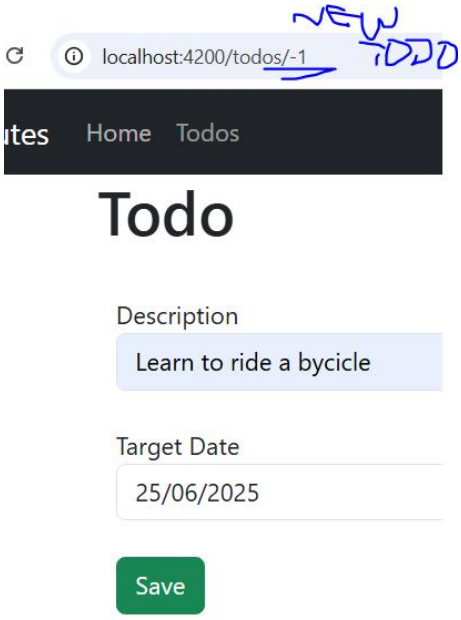

localhost:4200/todos
Home Todos
My Todo's
Delete of todo 1001 successfull
Description
TargetDate
Is Completed
Update
Delete
Learn Spring JPA
JUN 4, 2025
false
Update
Delete
Learn Microservices
JUN 4, 2025
false
Update
Delete
Add

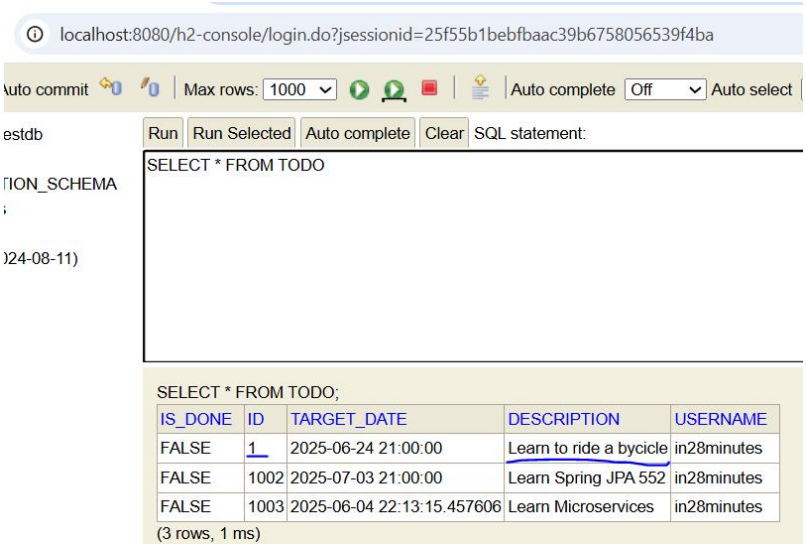

localhost:8080/h2-console/login.do?jsessionid=25f55b1bebfbaac39b6758056539f4ba
Auto commit
Max rows: 1000
Auto complete Off
:testdb
ATION_SCHEMA
2024-08-11)
Run
Run Selected
Auto complete
Clear
SQL statement:
SELECT * FROM TODO
SELECT * FROM TODO;
IS_DONE
ID
TARGET_DATE
DESCRIPTION
USERNAME
FALSE
1002
2025-06-04 22:13:15.457606
Learn Spring JPA
in28minutes
FALSE
1003
2025-06-04 22:13:15.457606
Learn Microservices
in28minutes
(2 rows, 0 ms)


localhost:4200/todos/1002
Home Todos
Todo
Description
NEW
Learn Spring JPA 552
Target Date
04/07/2025
Save


localhost:4200/todos
Home Todos
My Todo's
Description
TargetDate
Is Completed
Update
Delete
Learn Spring JPA 552
JUL 3, 2025
false
Update
Delete
Learn Microservices
JUN 4, 2025
false
Update
Delete
Add





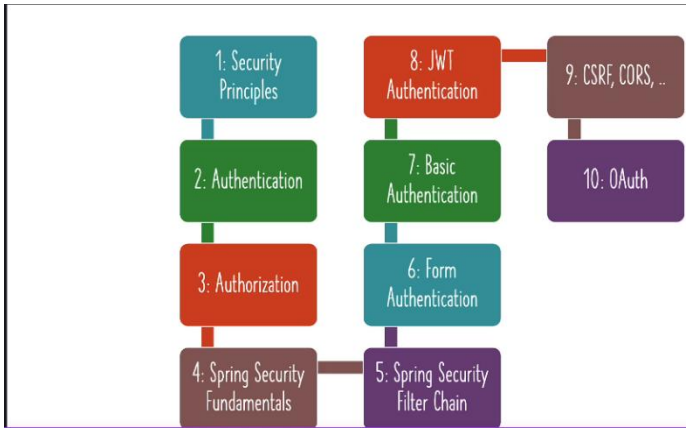


Now it is taking all of the todos, deleting, updating from database and inserting too.

```
RestfulWebServicesApplication [Java Application] C:\Program Files\eclipse-jee-2024-09-R-win32-x86_64\eclipse\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_21.0.4.v20240802-1551\jre\bin\javaw.exe  (4 de jun. de 2025 22:12:55) [pid: 11856]
Hibernate: delete from todo where id=?
Hibernate: select t1_0.id,t1_0.description,t1_0.is_done,t1_0.target_date,t1_0.username from todo t1_0 where t1_0.username=?
Hibernate: select t1_0.id,t1_0.description,t1_0.is_done,t1_0.target_date,t1_0.username from todo t1_0 where t1_0.id=?
Hibernate: select t1_0.id,t1_0.description,t1_0.is_done,t1_0.target_date,t1_0.username from todo t1_0 where t1_0.id=?
Hibernate: update todo set description=?,is_done=?,target_date=?,username=? where id=?
Hibernate: select t1_0.id,t1_0.description,t1_0.is_done,t1_0.target_date,t1_0.username from todo t1_0 where t1_0.username=?
Hibernate: select next value for todo_seq
Hibernate: insert into todo (description,is_done,target_date,username,id) values (?,?,?,?,?)
Hibernate: select t1_0.id,t1_0.description,t1_0.is_done,t1_0.target_date,t1_0.username from todo t1_0 where t1_0.username=?
```

Checking the terminal at the back-end side, it is possible to check the sql operations done by hibernate as a proof that the controller uris are being requested, processed and answered back.

From this point ahead is to depth the knowledge in JWT to explain about the spring security working, in the following steps:



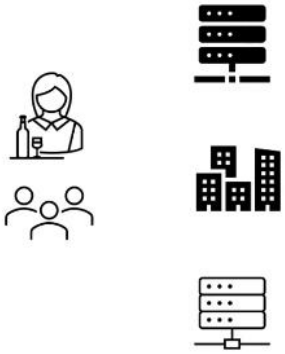
The security fundamentals are applied in any system, because it will guide the system working during the development.

Identifying the **resources** that the system will offer, a set of rest API services to be exposed, application (site) to consume the services, the database to store the client data (tables and relationships), a virtual machine on the cloud to run the app server.

Identifying the **identities / entities / actors** will access and manipulate the system using calls to modify the data and to set the allowed actions according to the profile..

Using the **authentication** to say who is the role / profile which is accessing the system by username and password that only the real user should remember and the ways to prove his identity (sms, email, key-pass).

Using the **authorization** to determine the level of hierarchy the user is subdue, for controlling the actions allowed to the type of user, for instance, the Adm can update and delete data while the common user can only retrieve the data.



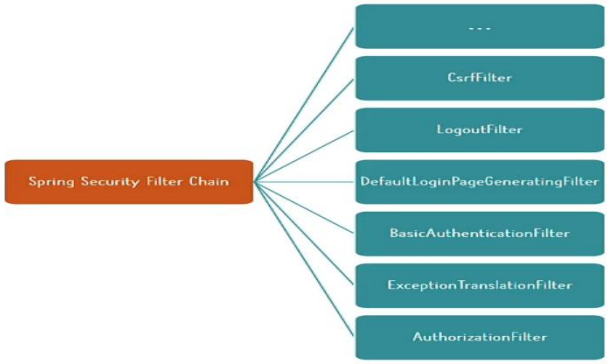
Every small flaw counts, so these principles will help to think in security:

1st Validate any request and its incoming content;

2nd Think about security requirements at beginning of the project considering which roles and privileges will be granted for each user as filters to be tested;

3rd Security since transport, network, infra, O.S and app built in a simpler way to be open to changes when needed;

Real life example: Get onboard in a plane, trust nothing, identify and check the **passenger** by **passport, id and the ticket,** the same way the **crew**, the **pilots**, the **check-in** team, **baggage inspection** team, **passengers** do not do **anything beyond** its limitations in every movement.



In Java, the spring security is the dependency focused in the protection of applications (rest API, Web apps, micro-services) covering a lot of points including authentication and authorization.

Obeying the principles, every each request and its data is intercepted by the spring security for applying a filter chain to allow or denying the continuation of process to reach a general controller (dispatcher Servlet) to analyze the request verb and url to specify which controller will be responsible for servicing the request.

Detailing the filters:

Authentication: is this user valid? -> AuthenticationFilter

Authorization: which right access does this user have? -> AuthorizationFilter

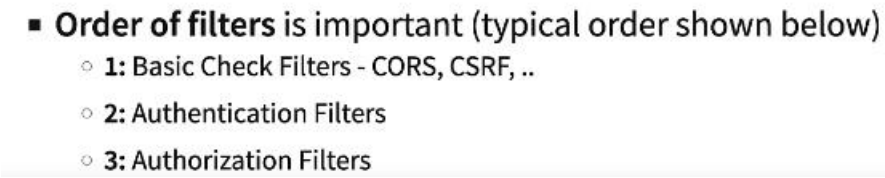
Cross-**O**rigin **R**esource **S**haring (CORS): Should a call from other domains be accepted?-> CorsFilter

Similarly, **C**ross-**S**ite **R**equest **F**orgery (CSRF): is the domain is legit? Or is it reusing some stolen client cookie?->csrfFilter

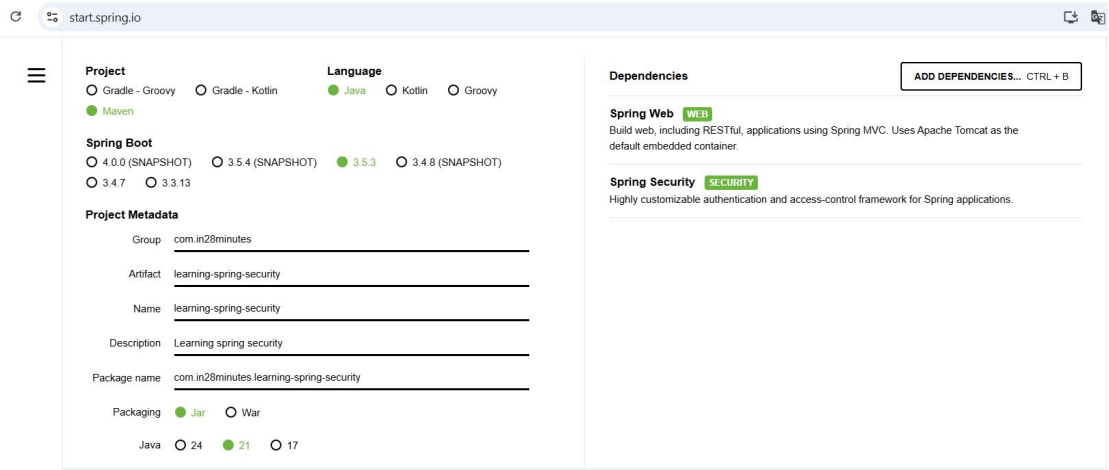
*Forgery -> fake

AutoGenerated Login&Logout page ->LogoutFilter,DefaultLoginPageGenerationFilter, DefaultLogoutPageGenerationFilter

In case of problems, exceptions, it's not cool to return the error as it happens on the back-end, it is needed to translate it to a language that the common user can understand->HTTP responses->ExceptionTranslationFilter (Entity Response -> 404 not found)



From this moment ahead, lets focusing on the hands on, open spring intlzr with these configs: Dependency manager Maven, Java language, spring boot latest version, java latest version and **dependecies as spring web and security**.

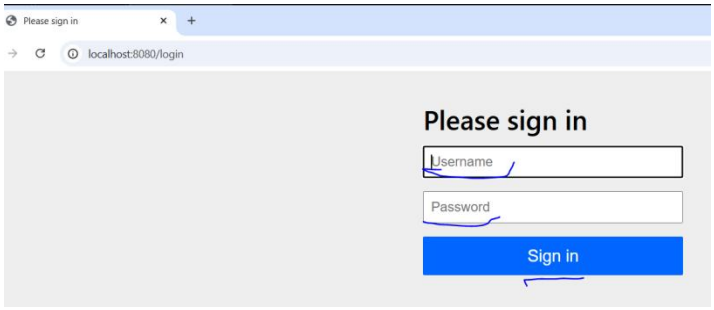


After importing the project and running on eclipse these info appear on the terminal showing the use of the dependencies:

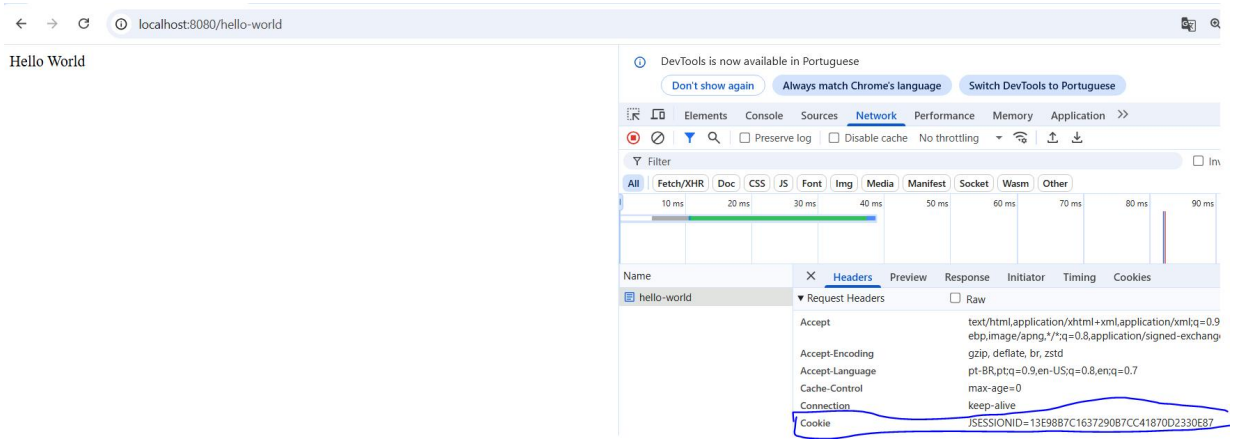
```
Using generated security password: a03129a9-94d5-4855-bd1f-99c3610112ba

This generated password is for development use only. Your security configuration must b

11-16 10:30:42 - o.s.s.web.DefaultSecurityFilterChain - Will secure any request with [o
```

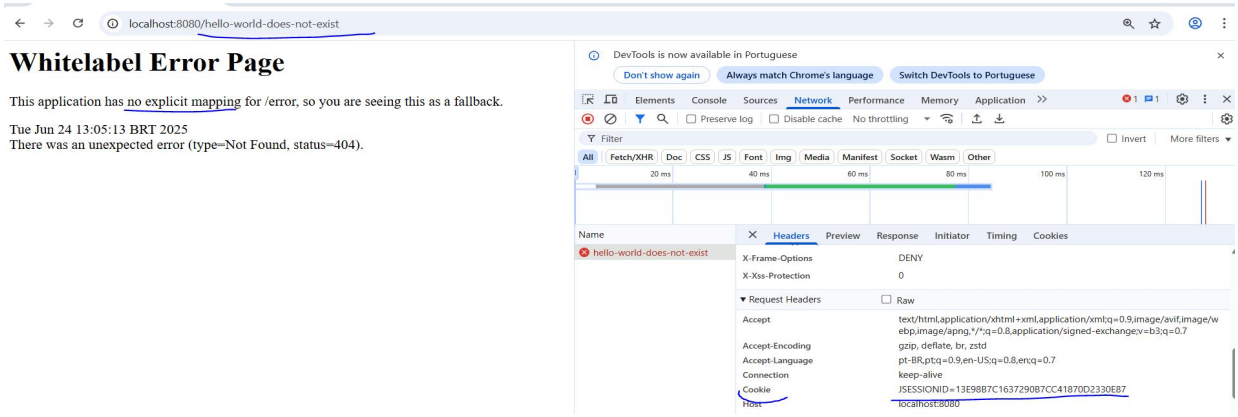
Using the spring security, when a service is available or not, the dependency protects the application from **every request** by default requiring the user is **authenticated** by **form** authentication (user+password) providing a login & logout page, once logged in, it is created a cookie property that contains a session Id that is sent in **every request to identify** the **authenticated user**. Then, when the user accesses the /logout route, a logout page is rendered asking if you really want to log out, confirming it, the user is redirected to the log in page with a reminder saying that the user has been logged out.

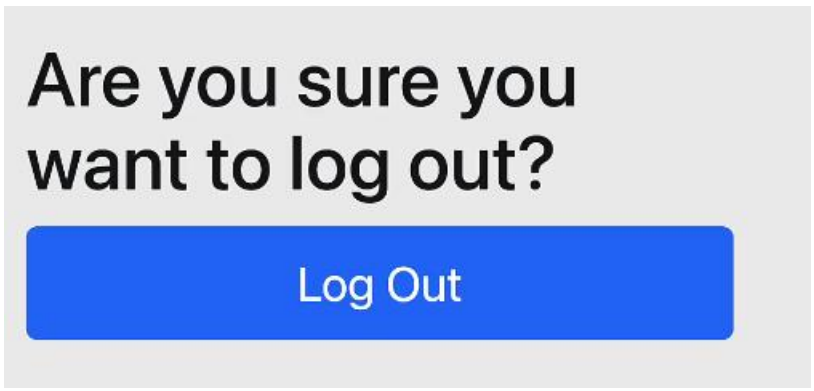
```java
@RestController
public class HellowWorldResource {

    @GetMapping("/hello-world")
    public String HellowWorld() {
        return "Hello World";
    }
}
```

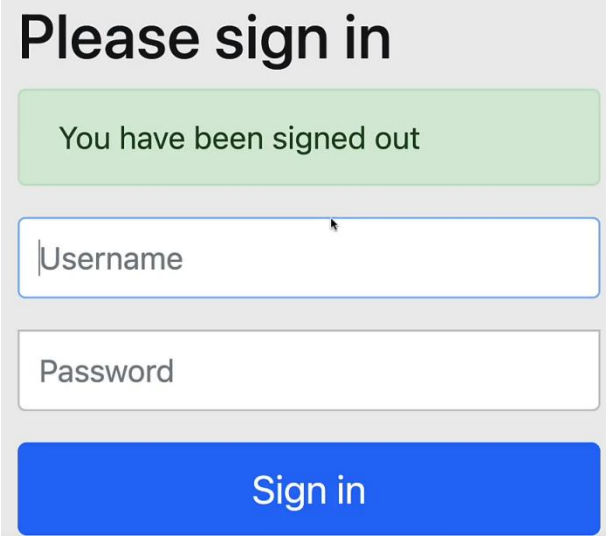


```
Using generated security password: 6acba7ba-804e-477a-8d2f-180426881c78
```

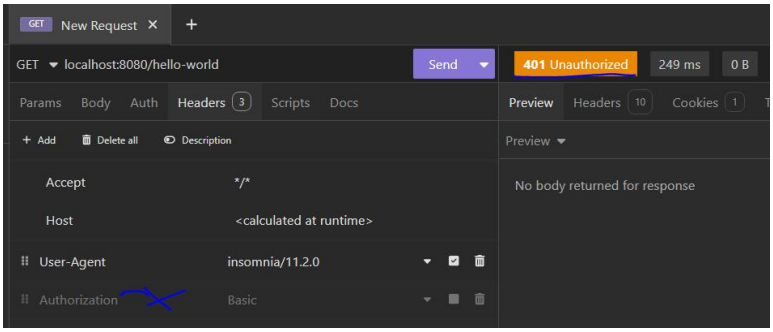


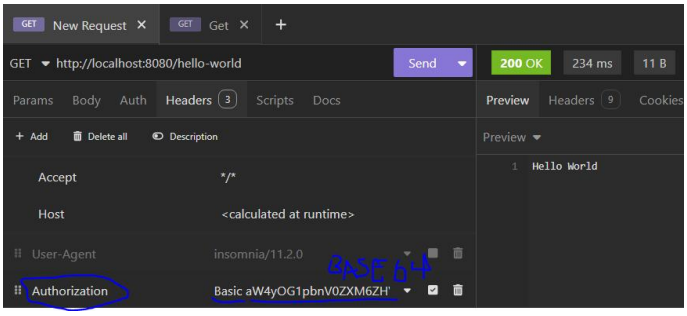






Another authentication way that is accepted by the spring security by default is the **basic** authentication as the most simple option that combines the username : password **encoded into base 64 along with 'Basic '** as a request header property called **authorization** in every request. But it is recommended for development purpose only, because it is constant if the user or password doesn't change and doesn't expire being easily decoded and not contains authorization data like privileges, roles, etc. . Anyway, the user needs to be authenticated for accessing the services.





To make the username and password static, on application.properties include the lines:

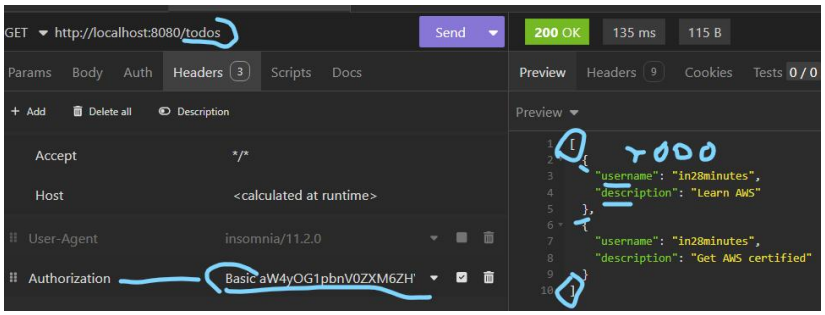
Spring.security.user.name=in28minutes

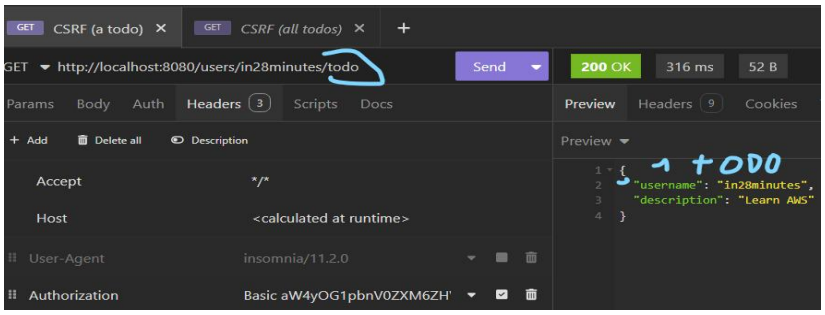
Spring.security.user.password=dummy

The next protection used by default in spring security is the **C**ross-**S**ite **R**equest **F**orgery (CSRF), if the cookie was not erased because the user didn't logged out, it will prevent the client cookie to be stolen and used in fraud actions by a malicious site.

- **2: SameSite cookie (`Set-Cookie: SameSite=Strict`)**
  - application.properties
    - `server.servlet.session.cookie.same-site=strict`

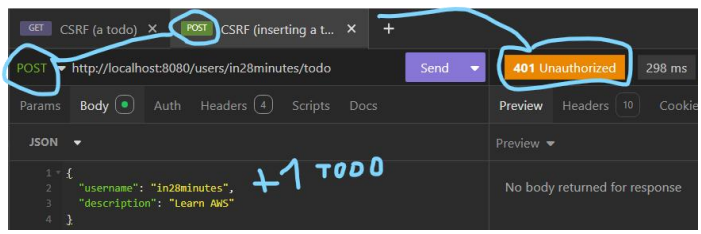
```java
@RestController
public class TodoResource {

    private Logger logger = LoggerFactory.getLogger(getClass());

    private static final List<Todo> TODO_LIST = List.of(new Todo("in28minutes","Learn AWS"),
            new Todo("in28minutes","Get AWS certified"));

    @GetMapping("/todos")
    public List<Todo> retrivelAllTodos() {
        return TODO_LIST;
    }

    @GetMapping("/users/{username}/todo")
    public Todo retrieveTodoForSpecificUser(@PathVariable String username){
        return TODO_LIST.get(0);
    }

    @PostMapping("/user/{username}/todo")
    public void createTodo(@PathVariable String username,
            @RequestBody Todo todo) {
        logger.info("created {} for {}", todo, username);
    }

}

record Todo (String username, String description) {}
```





Without doing any configuration, the practice is to create one more controller for retrieving todos from a static list containing two todos from a record, creating get routes to retrieve all of them, one of them for a specific username and simulate an update operation with a post route receiving a todo.

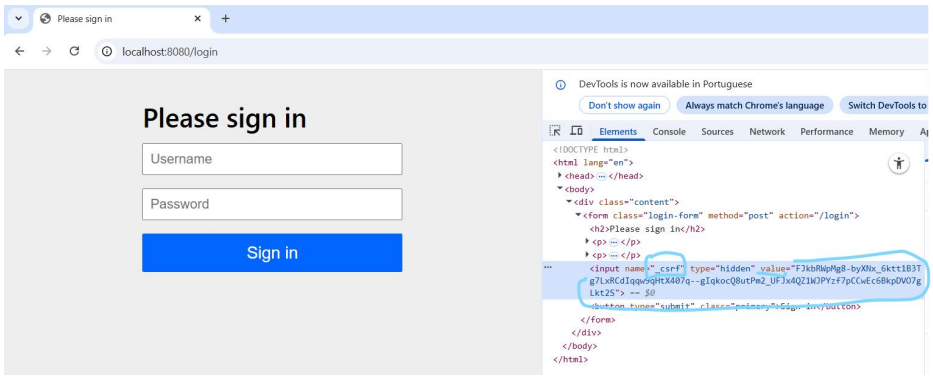
* record: it is a special type of class responsible for carrying a **plain** set immutable data. Being a functional way to create **plain** data classes without worrying with manual creation of constructors, getters,setters,equals,hashcode and to string (compiler is responsible for it).



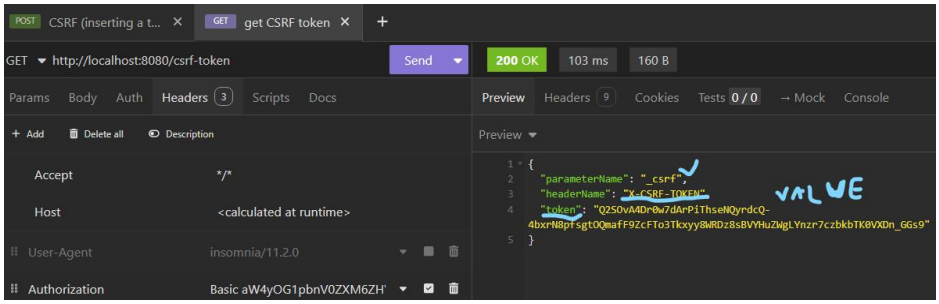
The solution provided is:

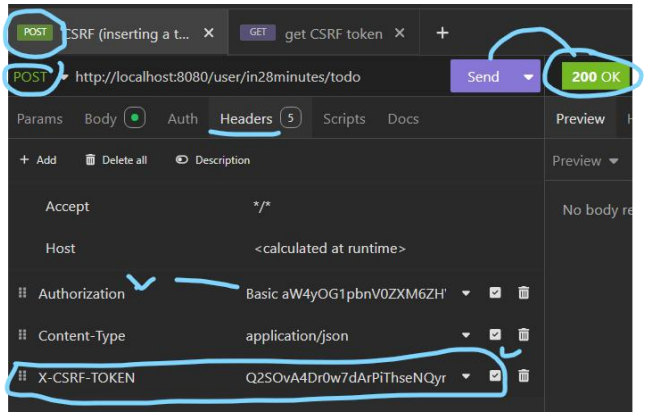
Creating a token for each request;

For updating data, CSRF token is needed from the previous request

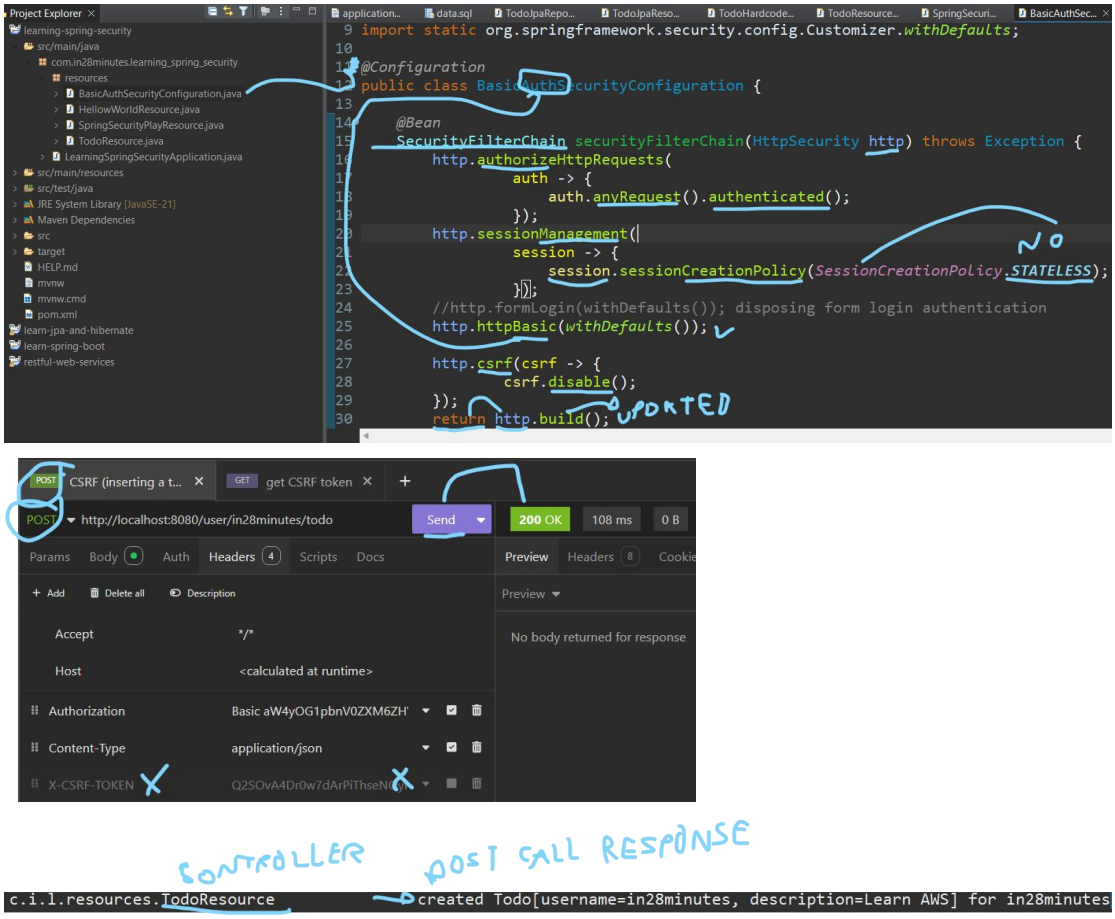



```java
package com.in28minutes.learning_spring_security.resources;

import org.springframework.security.web.csrf.CsrfToken;

@RestController
public class SpringSecurityPlayResource {

    @GetMapping("/csrf-token")
    public CsrfToken retrieveCsrfToken(HttpServletRequest request) {
        return (CsrfToken) request.getAttribute("_csrf");
    }
}
```





CSRF token is obtained by the request object from the attribute “_csrf” and cast it to a CSRF token interface. Then on insomnia, make the call to obtain the data: parameter name; headerName, token. With it on hands, it’s possible to make a post/put call.



To disable the csrf, it is needed to create a new configuration class like BasicAuthSecurityConfiguration with the annotation @Configuration, practically copying a pre-exist method from a class of security configuration in the spring boot dependencies, just commenting the formLogin that is the form authentication with login&logout pages, passing a customizer called withDefaults() as parameter on httpBasic that is the basic authentication and mainly including csrf disabling code, using the same http object calling the csrf method and call the disable method as parameter, as in, http.csrf(csrf-> csrf.disable()). This way, is possible to make a post call without using the recovered csrf token.

When a full-stack app is built, the back-end needs to be prepared to deal with front-end (outside) requests, thinking about it, the **C**ross-**O**rigin **R**esource **S**haring(CORS) allows to set which domains are allowed to make requests to the resources, that by default is blocked.
For specifying the domains, there are two ways:

Global:

```
@Bean
public WebMvcConfigurer corsConfigurer() {
    return new WebMvcConfigurer() {
        public void addCorsMappings(CorsRegistry registry) {
            registry.addMapping("/**")
                .allowedMethods("*")
                .allowedOrigins("http://localhost:3000");
        }
    };
}
```

The registry object can set free all the routes and http verbs to specific domains, having access to **all the controllers**.

Local: On the specific controller or method use the @annotation @CrossOrigin without specifying which domain or @CrossOrigin(origins="http://www.frontend.com").

Focusing on managing the users of the app instead of only one, there are 3 ways to do this:

In memory for development purpose only;

Database using JDBC/JPA to access the credentials;

LDAP - **L**ightweight **D**irectory **A**ccess **P**rotocol for accessing directory services and authentication.

Inside basicAuthSecurityConfiguration:

@Bean

Public UserDetailsService userDetailsService (){

Var user = User.withUsername("in28minutes")

.password("{noop}dummy")

.roles("USER")

.build();

Var admin = User.withUsername("admin")

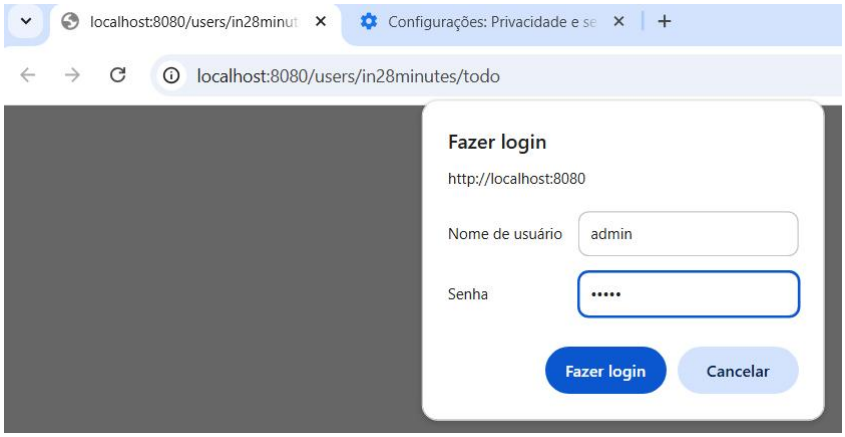
.password("{noop}admin")

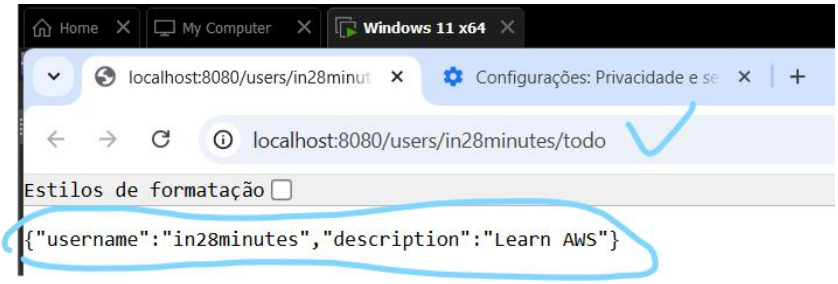
.roles("ADMIN");// I can include multiple roles, as in, 1 or +

.build();

Return new **InMemory**UserDetailsManager(user,admin);

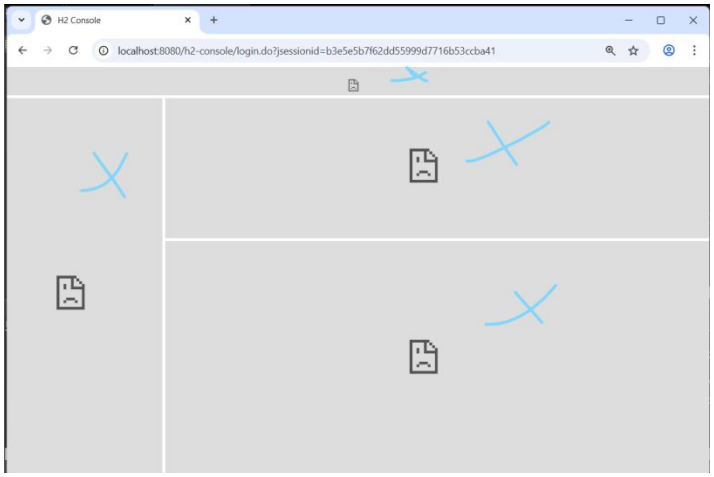
}






```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
</dependency>
```

```
spring.datasource.url=jdbc:h2:mem:testdb
```




```
http.headers(headers -> headers.frameOptions(
        frameOptionConfig -> frameOptionConfig.disable()));
```

The second experiment is to use a **database** called h2 to store the **user credentials.** Getting started by the dependency importing and the fixed url on app.properties. When we try to open it after connecting, a lot of frames are blocked by spring security, for sorting this out the frame option must disabled from the header.

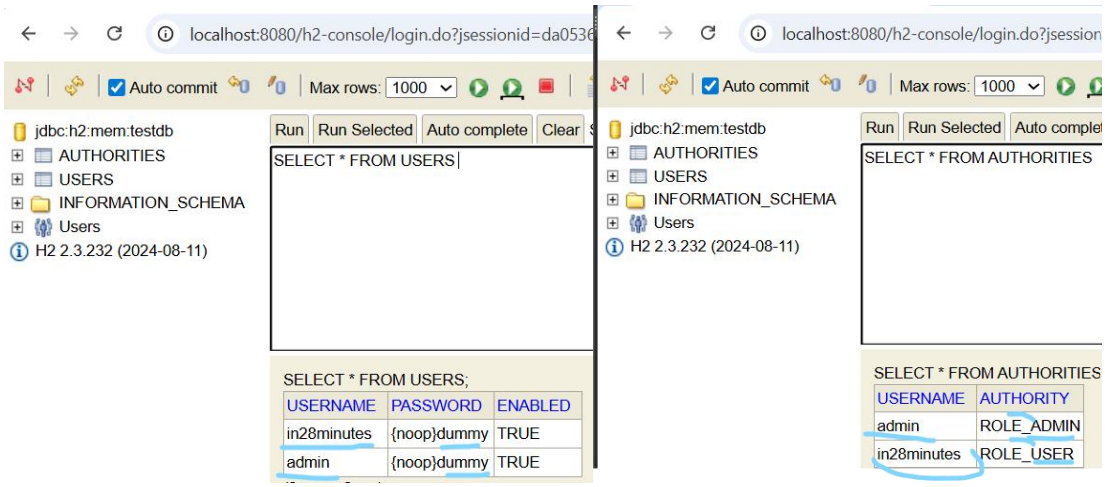
```
@Bean
public DataSource dataSource() {
        return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.H2)
             .addScript(JdbcDaoImpl.DEFAULT_USER_SCHEMA_DDL_LOCATION)
              .build();
}
```

Now that everything is configured, still on the configuration class, define which database is being used, in this case the embedded H2, so a new method called dataSource is created to be responsible for using the embedded database, choosing H2 as type, **and executing the sql file responsible for creating the users and authorities table.**

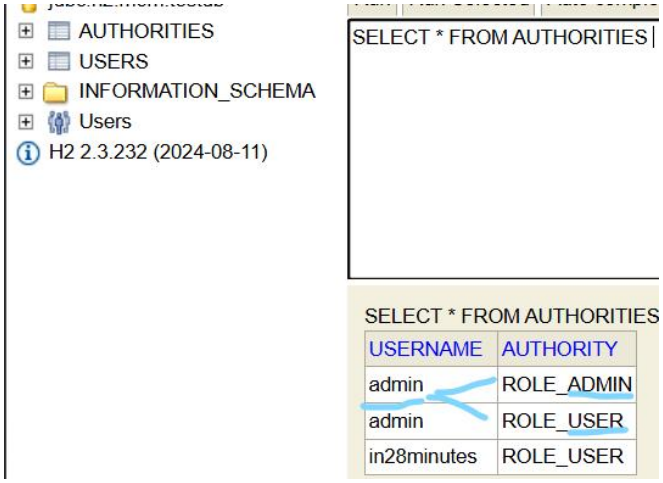
```
@Bean
	public UserDetailsService userDetailsService(DataSource dataSource) {
		var user = User.withUsername("in28minutes")
					.password("{noop}dummy")
					.roles("USER")
					.build();

		var admin = User.withUsername("admin")
					.password("{noop}dummy")
					.roles("ADMIN")
					.build();

		var jdbcUserDetailsManager = new JdbcUserDetailsManager(dataSource);
		jdbcUserDetailsManager.createUser(user);
		jdbcUserDetailsManager.createUser(admin);

		return jdbcUserDetailsManager;
}
```

Therefore, with the data source in hands, on the userDetailsService method make it as a parameter (dependency injection) for that after the **User** variables are created, instantiate the **jdbcUserDetailsManager** passing the data source as parameter to call the method **createUser** carrying the **user** variables as parameters and finally return it.

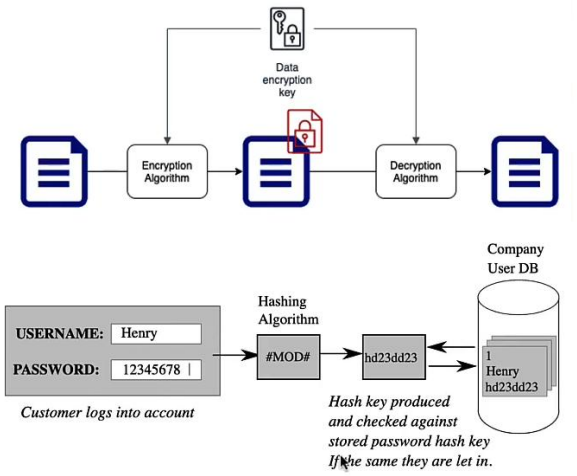
* {noop} -> no encoding.



```
var admin = User.withUsername("admin")
                .password("{noop}dummy")
                .roles("ADMIN","USER")
                .build();
```



The hands on result is the creation of the tables (user&authorities) with the **user** data object **in database not only memory.**



As a general overview, there are three data manipulation ways commonly used:

Encoding - Make a data to become another **different** one **without using** a key or password, being possible revert it to the original one. It is used in data **compression** or streaming, for instance: Base 64 (algorithm), Wav, MP3.

Encryption - Encoding & Decoding data **using a key or password to protect it.** Example: RSA.

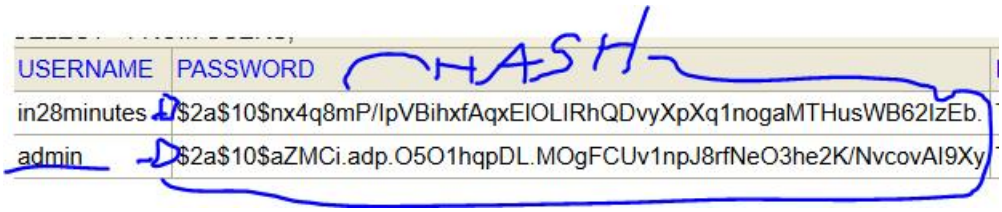
Hash - Convert data into a Hash/String during a one-way **not reversible** process to make sure the data integrity not returning to be the old value and yes a comparison object. Example: bcrypt, scrypt, argon 2.

The HASH algorithm that was used so far is SHA-256, but it became no longer secure due to hardware improving for brutal force. The recommendation is to use **adaptive one way**

functions with Work Factor of 1 second, as in, functions that turn data in hash with no return of the original value with data verification of 1 second.

On java, **the PasswordEncoder** is **the interface** that is implemented by several classes for this adaptation.

```
var admin = User.withUsername("admin")
                .password("dummy")
                .passwordEncoder(str -> passwordEncoder().encode(str))....
@Bean
Public BCryptPasswordEncoder passwordEncoder(){
        Return new BCryptPasswordEncoder();
}
```



To apply the encryption, create a method to return the BCryptPasswordEncoder object and after .password on the User builder, include passwordEncoder method passing the lambda expression defining the password reference used on password method -> use the passwordEncoder method created to take the instance and call the encode method passing the password reference. The result will be the table user having the simple “dummy” password as a hashing code sequence.

Detailing the JWT (Java Web Token), it is a **standard** in authentication and authorization of users to protect the data between two parties to avoid flaws, as features in comparison to basic auth, it has expiration time for invalidating stolen tokens, save user details and authorizations and not being easily decoded.

Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey
JzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpva
G4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKx
wRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

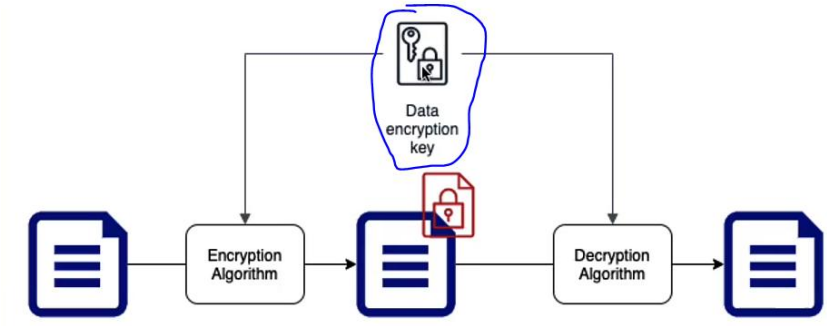
PAYLOAD: DATA

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

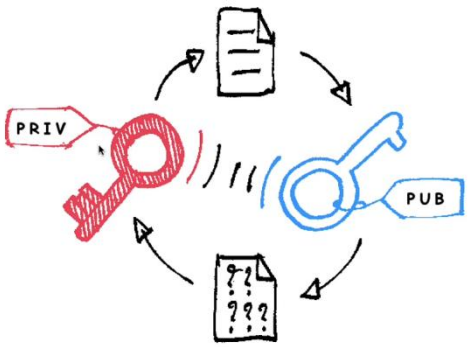
VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) secret base64 encoded
```

The JWT structure is header, payload and signature. The header contains the hash algorithm, type of token, the payload contains the data as iss: the issuer, sub: the subject, aud: the audience, exp: expiration time, iat: token creation date and custom attributes, the signature having the header and payload encoded and the **secret key.**



This key can be symmetric or asymmetric. The symmetric uses the **same key for encryption and decryption of the data**, choosing the right algorithm and securely sharing the key.

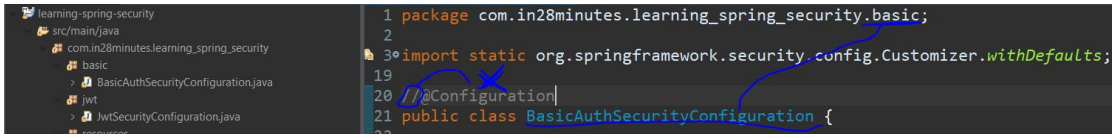


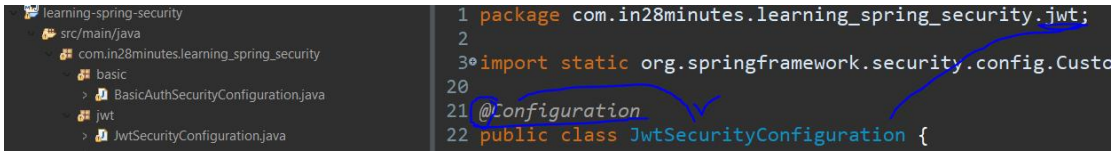
The asymmetric key is composed by **TWO keys, public and private keys,** where the public key is shared with **trusted** everybody, encrypts the simple data and the private key shared **only with you,** decrypts the encoded data. So, one key doesn't finds another, just for encryption or decryption, being the asymmetric key the recommended for business.

The flow is create a JWT for encoding user credentials, data and RSA key, send the JWT on header request as Authorization: Bearer ${JWT_TOKEN} to be decoded and verified using the RSA key.

- **1:** Create Key Pair
  - We will use `java.security.KeyPairGenerator`
  - You can use openssl as well
- **2:** Create RSA Key object using Key Pair
  - `com.nimbusds.jose.jwk.RSAKey`
- **3:** Create JWKSource (JSON Web Key source)
  - Create `JWKSet` (a new JSON Web Key set) with the RSA Key
  - Create `JWKSource` using the `JWKSet`
- **4:** Use RSA Public Key for Decoding
  - `NimbusJwtDecoder.withPublicKey(rsaKey().toRSAPublicKey()).build()`
- **5:** Use JWKSource for Encoding
  - `return new NimbusJwtEncoder(jwkSource());`


```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
```






```
http.oauth2ResourceServer(
        OAuth2ResourceServerConfigurer -> {
            OAuth2ResourceServerConfigurer.jwt(Customizer.withDefaults());
        });
```

For implementing the JWT on the project is needed a lot of steps, but for getting started, include the dependency **oauth2-resource-server** on the pom.xml file, reuse the

BasicAuthenticationSC code on the new JwtSecurityConfiguration class, separate them in different packages, then disable the BasicAuthSC commenting the @Configuration annotation. Just an observation, the same way the httpBasic and formLogin enable specific ways to authenticate the user, the method **oauth2ResourceServer** is responsible for enabling the **jwt authentication** like an **authentication provider**.

```
@Bean
public KeyPair keyPair() {
    try {
        var keyPairGenerator = KeyPairGenerator.getInstance("RSA");
        keyPairGenerator.initialize(2048);
        return keyPairGenerator.generateKeyPair();
    }catch(Exception ex) {
        throw new RuntimeException(ex);
    }

}
```

The first step is to create the **two keys** through the **KeyPair object**, using the class KeyPairGenerator to create the public and private keys using the algorithm passed as string, determine the key size bigger is better, then return the keys if by chance the algorithm doesn't exist the exception is thrown.

```
@Bean
public RSAKey rsaKey(KeyPair keyPair) {
        return new RSAKey
                .Builder((RSAPublicKey) keyPair.getPublic())
                .privateKey(keyPair.getPrivate())
                .keyID(UUID.randomUUID().toString())
                .build();
}
```

After the two keys are created, an object **RSAKey** needs to be created containing the public key, private key, and the own ID that in this case is a random one.

```
@Bean
public JWKSource<SecurityContext> jwkSource(RSAKey rsaKey) {
    var jwkSet = new JWKSet(rsaKey);
    return (jwkSelector, context) -> jwkSelector.select(jwkSet);
}
```

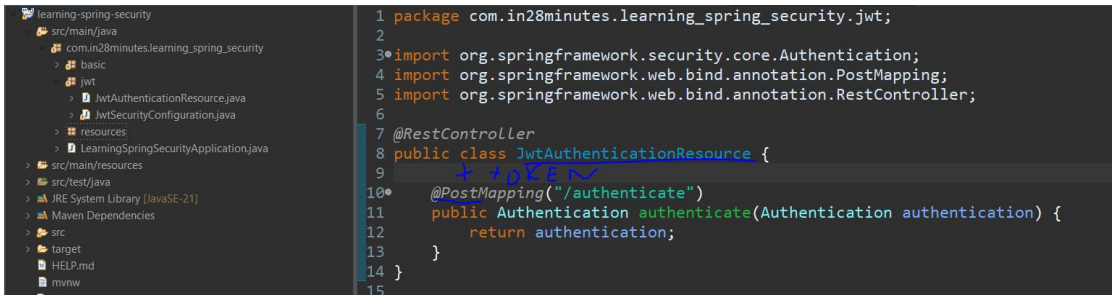
Also is needed to create JSON Web Key Source using a jwkSet with the rsaKey, being a way to store the cryptography keys in JSON format of the algorithm RSA **to validate the JWT token signatures**.

```
@Bean
public JwtDecoder jwtDecoder(RSAKey rsaKey) throws JOSEException{
    return NimbusJwtDecoder
            .withPublicKey(rsaKey.toRSAPublicKey())
            .build();
}
```

After JWKSet and source are defined, it is needed to decode the data using the public key in rsa object to return a **JwtDecoder implementation**.

```
@Bean
public JwtEncoder jwtEncoder(JWKSource<SecurityContext> jwkSource) {
    return new NimbusJwtEncoder(jwkSource);
}
```

The same way the decoder is needed, the encoder is too, then the **jwtEncoder implementation object** using jwkSource as parameter is returned.



Once the configuration is done, it is needed to start building the controller to supply the jwtToken for making any other request. The Authentication object will be used to obtain the token.

```
Private JwtEncoder jwtEncoder;
Public JwtAuthenticationResource(JwtEncoder jwtEncoder){
        This.jwtEncoder = jwtEncoder;
}
```

Based on the configuration, the jwtEncoder has the **algorithm, the private and public keys**, this way, declaring a new **reference** on rest controller constructor **will carry all this info**.

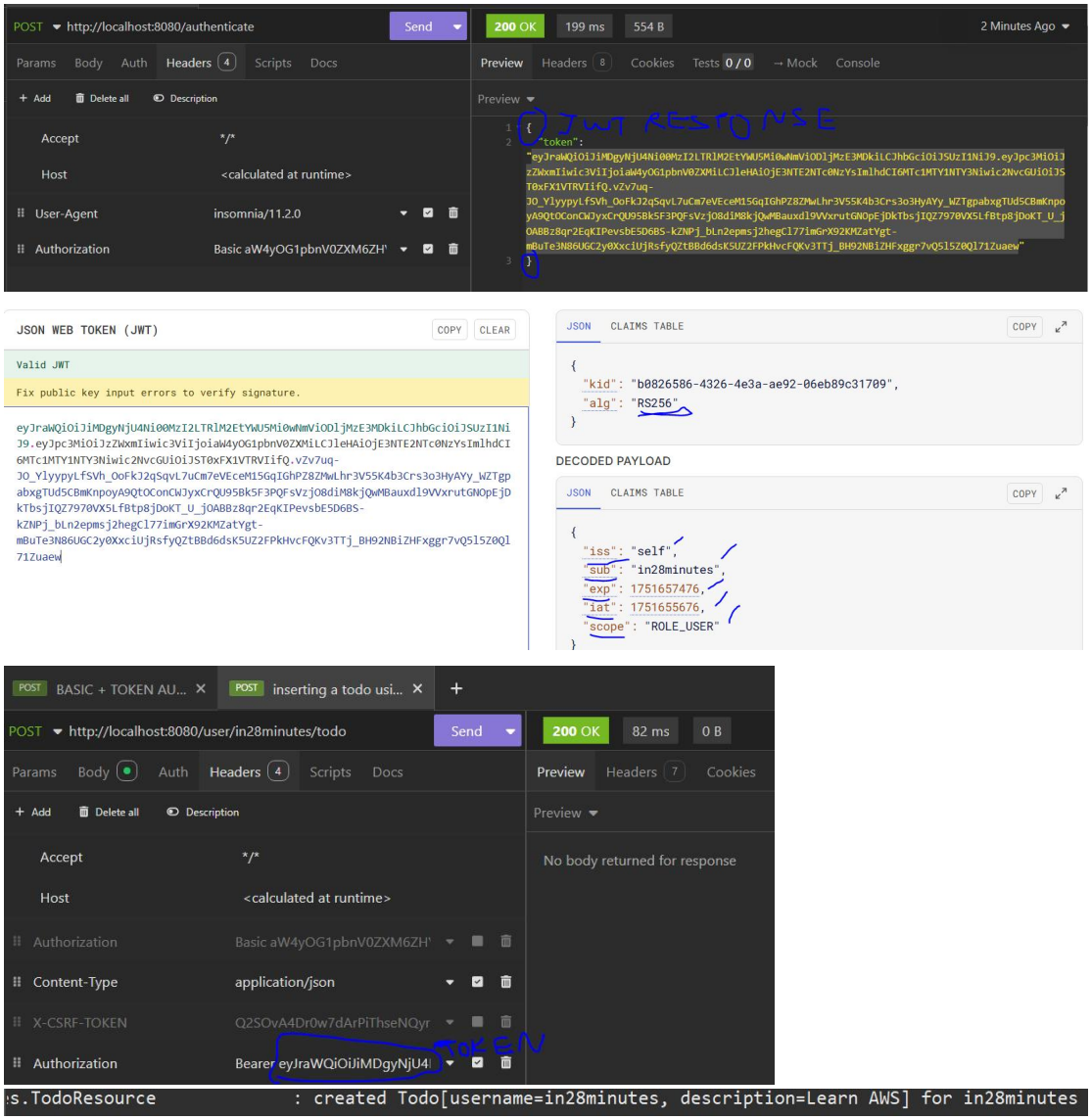
```
....
Public JwtResponse authentication(Authentication authentication){
Return new JwtResponse(createToken(authentication));
}
Private String createToken(Authentication authentication){
Var claims = JwtClaimsSet
                        .builder()
                        .issuer(“self”)
                        .issuedAt(Instant.now());
                        .expiresAt(Instant.now().plusSeconds(60L * 30L);
                        .subject(authentication.getName())
                        .claims(“scope”, createScope(authentication))
                        .build();
Return JwtEncoder.encode(JwtEncoderParameters.from(claims)).getTokenValue();
}
Record JwtResponse (String token){}
```

But it doesn’t have **the payload,** so thinking about it, a method is created to this using the authentication object as part of creation of the payload and token. This creation is encapsulated on the object **JwtClaimsSet** to define: issuer(creator),issuedAt(current time),expiresAt(current time + 30 minutes),subject(Username from auth),claims(scope of authorization from auth inside another new method).

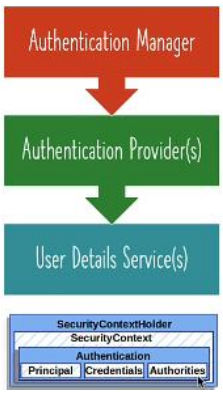
Now the **payload** is filled in, use the jwtEncoder method to encode **it** using the class JwtEncoderParameters with the payload, to finally obtain the validated token by JWKs in a record called JwtResponse as return of the request.

```
Private String createScope(Authentication authentication){
        Return authentication
                .getAuthorities()
                .stream()
                .map(a -> a.getAuthority())
                .collect(Collectors.joining(“ ”));
}
```

This scope comes from a list of authorities, because the user can have 1 or more roles that needs to be reduced to a String like “ADMIN USER”.



Making the request to this authentication service, the token is obtained.



The Authentication in depth is the use of authentication manager to intercept the requests and validate them using the authentication provider (JWT) to obtain the credentials(username&password).

```
@Bean
        public UserDetailsService userDetailsService(DataSource dataSource) {
                var user = User.withUsername("in28minutes")
                                        .password("dummy")
                                        .passwordEncoder(str -> passwordEncoder().encode(str))
                                        .roles("USER")
                                        .build();

                var admin = User.withUsername("admin")
                                         .password("dummy")
                                        .passwordEncoder(str -> passwordEncoder().encode(str))
                                         .roles("ADMIN")
                                         .build();
                ....
```

Once validated, the user data known as principal and authorities(roles) are retrieved from UserDetailsService.

The data are stored in Authentication(principal) and GrantedAuthority(roles).

```
//@Configuration
public class BasicAuthSecurityConfiguration {

    @Bean
    SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http.authorizeHttpRequests(
                auth -> {
                    auth
                    .requestMatchers("/users").hasRole("USER")
                    .requestMatchers("/admin/**").hasRole("ADMIN")
                    .anyRequest().authenticated();
                });
```

Specially about authorization, the global and method one can applied. For **global** is through the method **requestMatchers** to specify the user role allowed to do some request.

```java
@GetMapping("/users/{username}/todo")
@PreAuthorize("hasRole('USER') and #username == authentication.name")
@PostAuthorize("returnObject.username == 'in28Minutes'")
@RolesAllowed({"ADMIN", "USER"})
@Secured({"ROLE_ADMIN", "ROLE_USER"})
public Todo retrieveTodoForSpecificUser(@PathVariable String username){
    return TODO_LIST.get(0);
}
```

```java
@EnableMethodSecurity(jsr250Enabled = true, securedEnabled = true)
public class TodoResource {

    private Logger logger = LoggerFactory.getLogger(getClass());

    private static final List<Todo> TODO_LIST = List.of(
            new Todo("in28minutes","Learn AWS"),
            new Todo("in28minutes","Get AWS certified"));
```

In **method security** there multiple options to protect the access to a method, including:

@PreAuthorize-> **before** making the request, it is verified if the user matches some conditions for example, role and username (pathVariable = authentication).

@PostAuthorize-> **after** making the request, it is verified the user matches some conditions for example, if the username of the return object is 'x'.

@RolesAllowed({"",""})-> check **the roles allowed** to request the service.

@Secured({"ROLE_..."},{"ROLE_..."})-> check **the authorities** to request the service

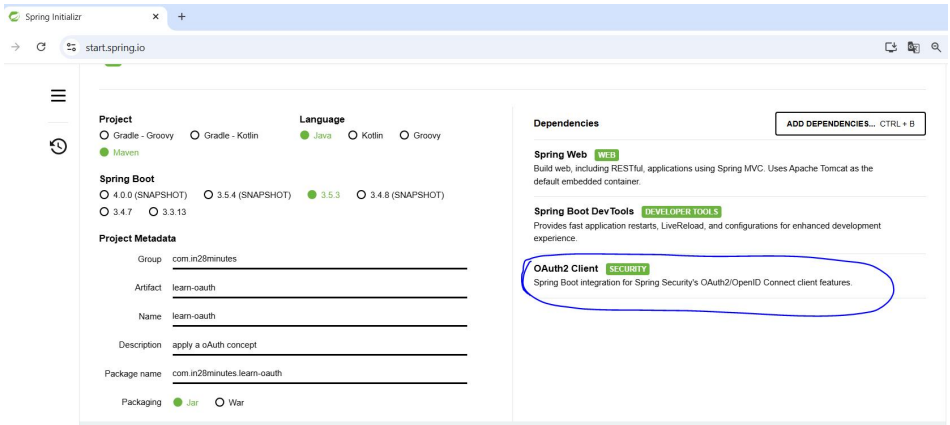
As a fullfilment of the OAuth, lets suppose that we want to share a google drive file with the todo management app. Then, there are some roles that can be identified:

Resource owner: ourselves.

Client App: Todo Management app.

Resource server: contains the resources that are being accessed- Google Drive

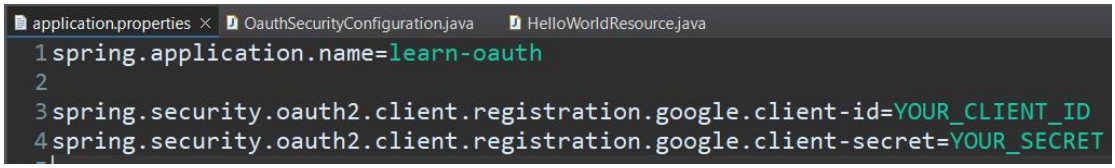
Authorization Server: Google OAuth Server



As a hands-on simulation, a new spring boot project is created with the **OAuth2 Client** dependency to make the bridge between the project and google.

Setting up the Web Security Configuration, a new file is created with the @Configuration annotation and the following method:

```
@Configuration
Public ...
SecurityFilterChain defaultSecurityFilterChain(httpSecurity http) throws Exception {
        http.authorizeHttpRequests(
                Auth -> {
                Auth.anyRequest().authenticate();
                })
                //http.formLogin();
                //http.httpBasic();
                http.oauth2Login(Customizer.withDefaults());
                Return http.build();
}
...
```



Configuring the application.properties file to store the google client id & secret to **authorize the google user** to access the project resource instead of **default formLogin and basic auth** by spring security Oauth2. For obtaining these info, it is needed to access the website https://console.cloud.google.com/ , and follow these steps:

° login with your google account

°on the dashboard:

° create credentials-> OAuth client ID

° if it is the first time, it is needed to create an **OAuth consent** to authorize the external access to the resource. Because the google authentication will **share client information with our project** like profile info and email (scope), then it is needed to configure our project presentation to the user while authentication like app name, support email, app logo,

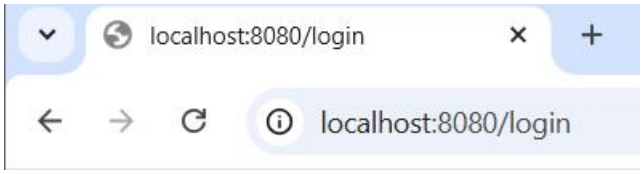
°After the OAuth consent is created, create credentials-> OAuth client ID

° Define the app kind, name and a default uri(s) before reach our project route like http://localhost:8080/login/oauth2/code/google and press create.

° A pop-up window is displayed containing the client-id and secret, copy them and replace YOUR_CLIENT_ID & YOUR_SECRET at application.properties.

application.properties | OauthSecurityConfiguration.java | HelloWorldResource.java

```java
package com.in28minutes.learn_oauth;

import org.springframework.security.core.Authentication;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloWorldResource {

    @GetMapping("/")
    public String helloWorld(Authentication authentication) {
        System.out.println(authentication.getPrincipal());
        System.out.println(authentication.getCredentials());
        System.out.println(authentication.getDetails());
        return "Hello World";
    }
}
```



Login with OAuth 2.0

Google

Hello World

After this configuration, we are able to create the controller to return at least a “hello-world” and on the terminal to access the user details through the authentication object.

Course link: Course: Go Java Full Stack with Spring Boot and Angular | Udemy

Course Repository: https://github.com/in28minutes/full-stack-with-angular-and-spring-boot

Extras:

Learn Python

THE BEST PYTHON COURSE EVER

Option: Learn Cloud

1: EASIEST AZURE COURSE

2: EASIEST AWS COURSE