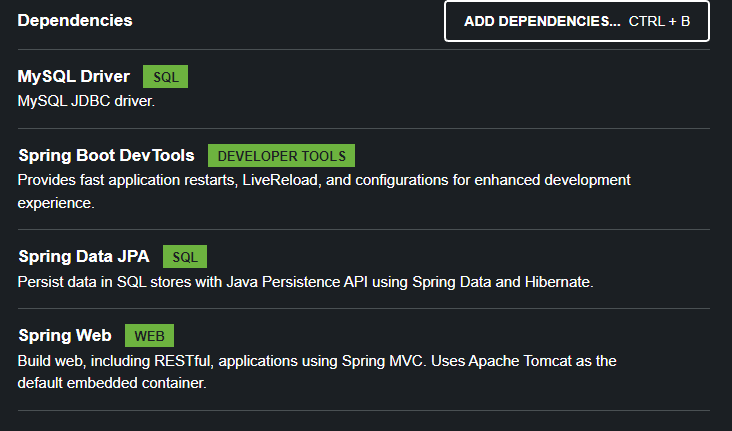


1 - Creamos el proyecto desde la web de Sprinboot y agregamos las dependencias que aparecen en -pantalla

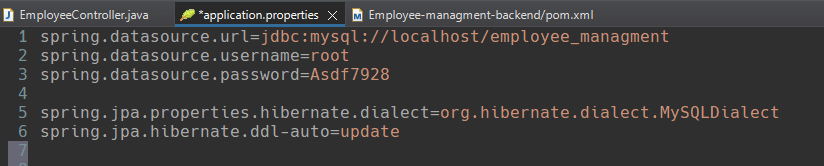
2 - Desde MYSQL vamos a crear la base de datos y comenzar a utilizarla con los siguientes comandos

Create database “nombre de la db”

Use “nombre de la db”

(En este caso sería “create database employee managment”)

3 - En Eclipse vamos a src/main/resources > application.properties y vamos a configurar la DB



\*1 – Es la dirección de url a la base de datos. En este caso es localhost y la siguiente sentencia es el nombre de la db

\*2 – Creería que es el nombre del user de MYSQL, lo estoy probando

\*3 – Mismo

\*5 – Hace referencia al dialecto utilizado  (Hay que revisar que sea compatible con la version deMySQL)

\*6 – Hace que cuando se abra la app lo primero que haga sea el update

4 – En la carpeta src/main/java creamos el paquete MODEL para crear las clases a utilizar. En este caso Employee   
A la clase le agregamos las anotaciones @Entity y @Table (name = “employee”). Importamos ambas anotaciones (en name = employee es para marcar a que tabla de la DB va a hacer referencia)   
Le agregamos ya dentro de la clase:

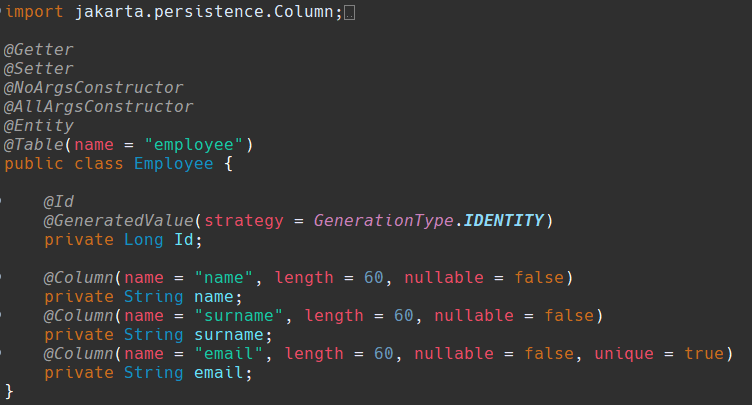
@Id   
@GeneratedValue(strategy = GenerationType.IDENTITY)

El @Id es para marcar que va a ser la primary key y el @GeneratedValue es para hacer que se auto incremente

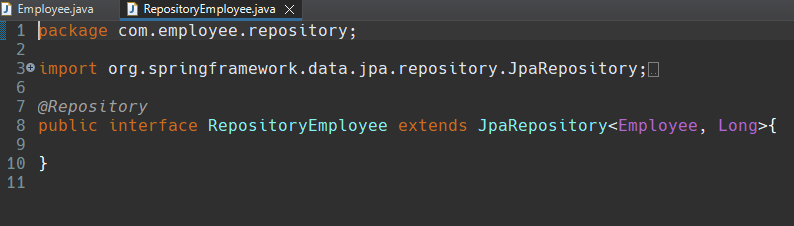
Luego creamos los atributos privados (a estos le podemos agregar “largo” de caracteres y si es nulleable o no, y, en caso del mail lo podemos settear como “unique” para que no pueda repetirse en la bd) e importamos la dependencia lombok para poder utilizar las anotaciones de getter y setter (las anotaciones de los getters y setters tienen que estar por encima de la public class Employee.

(Ademas de agregar la dependencia, al ser un Proyecto que se ejecuta en Spring, hay que instalar Lombok)

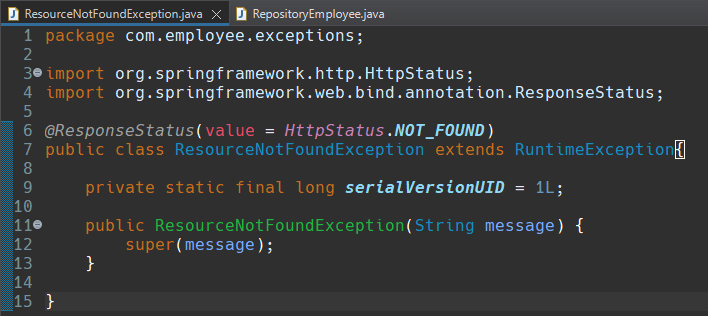
<**dependency**>   
 <**groupId**>org.projectlombok</**groupId**>   
 <**artifactId**>lombok</**artifactId**>    
 <**scope**>provided</**scope**>    
</**dependency**>



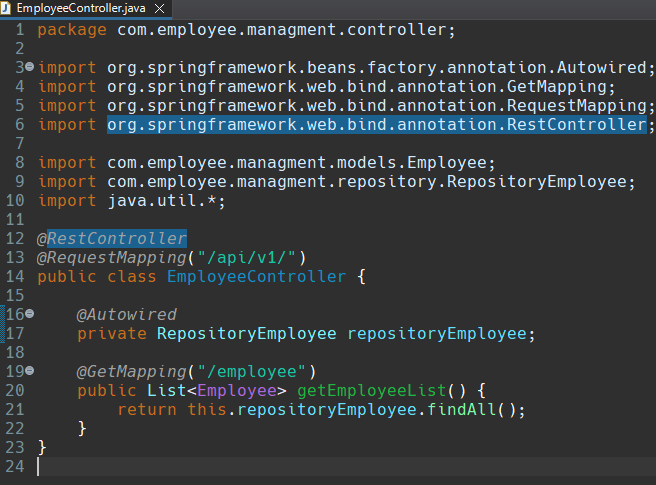
5 – En la carpeta src/main/java creamos el paquete com.employee.repository y creamos la interfaz RepositoryEmployee (esta se encarga de gestionar todas las operaciones de persistencia contra una tabla de la base de datos, es decir, es la que va a guardar y recuperar la información de la base de datos, por eso al extenderla de JpaRepository hay que pasar en los diamantes, la <clase> y el tipo de <primary key>)   
A la interfaz, le agregamos la anotacion de @Repository   
Luego le agregamos “extends JpaRepository<Employee, Long>”   
Employee va a ser la clase que trabaja y Long es el ID o key   
\*Importamos la herencia de Jpa



6 – Ahora vamos a crear el paquete para la excepciones “com.employee.exceptions” (tambien  dentro de main/java)   
Creamos la clase ResourceNotFoundException   
Le agregamos la anotación @ResponseStatus(value HttpStatus.NOT\_FOUND) y le agregamos la extension de RuntimeException   
Creamos un atributo privado static final long “serialVersionUID = 1L;”  (esto le agrega el numero de version a la clase creada… tengo que investigar mas sobre esto)  
Luego creamos el constructor que recibe un String “message” desde la clase super



7 – Creamos el paquete controller y adentro la clase EmployeeController.  (Controller es la clase que va a marcar las URL por las que el usuario puede interactuar con la app, va a llevar los metodos get, set, post, delete, etc)  
Le agregamos la anotación @RestController y @RequestMapping(“/api/v1/”) 🡨 esta es la URL que vamos a utilizar para acceder a la app; en este caso se usa apiv1 pero podría ser cualquier otra  
Creamos un atributo del tipo RepositoryEmployee y le agregamos la anotación @Autowired (inyección) para que sepa que esa información va a ser inyectada desde la bd o desde la app.  
Luego creamos el metodo getEmployeeList() que retorna repositoryEmployee.findAll(); esto nos va a devolver la lista completa de empleados. Se le agrega la anotación @GetMapping(“/employees”, en la imagen está sin la S pero la forma correcta de escribir empleados es con S) para asignarle una url y que en ella nos muestre el retorno (en este caso es la lista de empleados)



* + EN ESTA INSTANCIA YA PODEMOS PROBAR EL FUNCIONAMIENTO DE LA APP  
    VAMOS A MYSQL Y AGREGAMOS MANUALMENTE DOS EMPLEADOS Y DESDE LA URL <http://localhost:8080/api/v1/employee>s VAMOS A VER LOS RESULTADOS

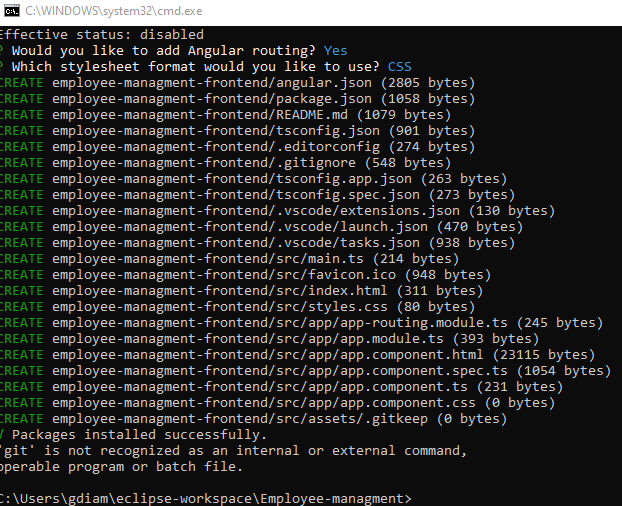
8 – Ahora vamos a crear la carpeta del FRONTEND

Primero debemos instalar angular, para hacerlo, abrimos el CMD (commandos) y parados sobre la ruta del entorno de trabajo de eclipse (para esto escribimos “cd C:\Users\gdiam\eclipse-workspace\Employee-management”) y escribimos “npm install -g @angular/cli”

(También podremos actualizer la version de angular si es que lo deseamos)

Luego de hacer esto escribimos

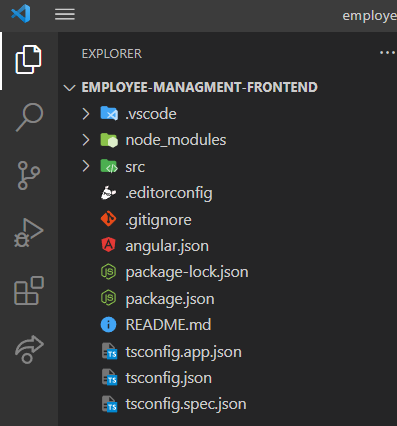
ng new employee-management-frontend y le damos enter, luego si pregunta algo de google le damos no, cuando pregunte si queremos agregar la ruta de angular ponemos que si y por ultimo al momento de elegir el stylesheet elegimos CSS



Una vez terminada la creación de la carpeta, vamos a vistual studio y hacemos

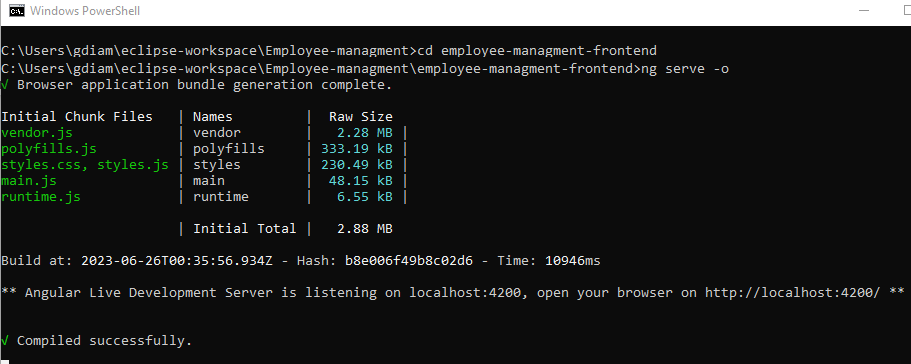
“file > open file > “ y buscamos la ruta donde se encuentra el Proyecto del frontend

C:\Users\gdiam\eclipse-workspace\Employee-managment\employee-managment-frontend



Ahora cambiamos la ruta del CMD y nos paramos dentro de la carpeta de frontend (cd employee-management-frontend) y escribimos “ng server -o” para ejecutar la aplicación (levantar el server)

(para acceder a la misma hay que ir a localhost:4200 desde un navegador)



En VS vamos a source > app > app.component.html y borramos todo el codigo que allí se encuentra

Vamos a crear el front desde 0. Para empezar, vamos <https://getbootstrap.com/docs/4.0/migration/> luego en la barra de la izquierda seleccionamos “Getting Started” y copiamos el link de CSS

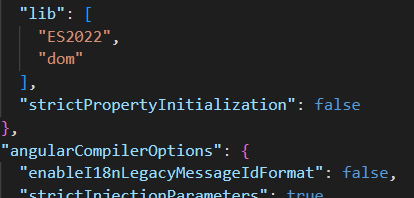
<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@4.0.0/dist/css/bootstrap.min.css" integrity="sha384-Gn5384xqQ1aoWXA+058RXPxPg6fy4IWvTNh0E263XmFcJlSAwiGgFAW/dAiS6JXm" crossorigin="anonymous">

Este Código lo vamos a pegar en el body del index.html

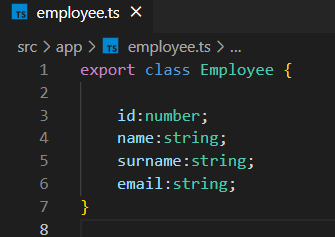
10 – Desde VS abrimos una terminal de powershell y tipeamos “ng g class employee” con esto Podemos crear la clase dentro de la app

(en caso de que haya un error para correr los scripts, tenemos que abrir como administrador desde windows el cmd de powershell y tipear lo siguiente “Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope LocalMachine”. Luego le damos Y y volvemos a correr el commando y le damos A y listo)

Ahora el archive tsconfig.json agregamos “strictPropertyInitialization” : false

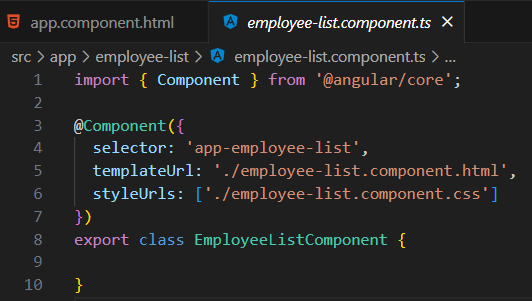


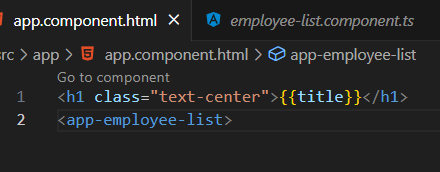
Y ahora en la clase Employee escribimos los atributos



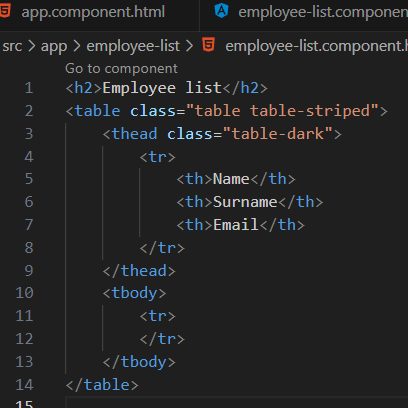
11 – Desde la terminal tipeamos “ng g c employee-list” para crear el componente del mismo nombre (la c es de component) que aparecera como una carpeta dentro de la carpeta “app”

Ahora, en el archive employee-list.component.ts copiamos el link que figura en selector y lo pegamos en el app.component.html





Vamos a Employee-list.component.html y creamos una table donde apareceran los datos del empleado.



12 – Ahora vamos al archivos employee-list-component.ts y en la clase EmployeeListComponent creamos una lista vacía (employees) y la llenamos con 2 empleados hardcodeados dentro de ngOnInit() que corre cuando se levanta la app



13 – Volvemos a employee-list.component.html y utilizamos el dialect de angular para llamar a la lista y que complete los campos con los datos de cada element de la misma

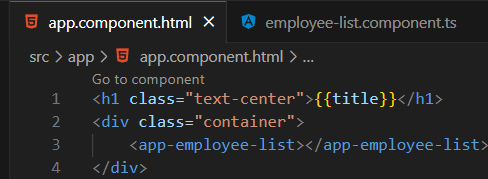


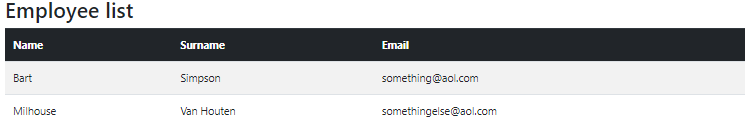
(notar las palabras clave “\*ngFor” que corre un foreach, “let” para crear la variable employee “of” employees)



Ahora ya Podemos ver la lista completada con los datos hardcodeados en la url http//localhost:4200

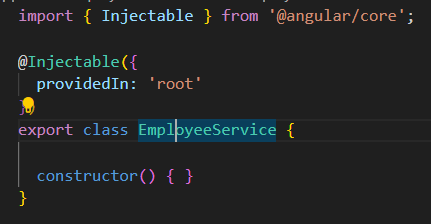
Aprovechamos la oportunidad y en app.component.html acomodamos el Código para que se vea major



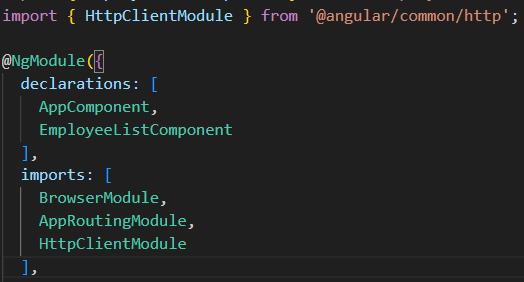


14 – Vamos a crear el servicio employee, para esto escribimos “ng g s employee” (s de service)

(hay que corroborar que en el archive employee.service.ts el servicio tenga la anotacion @injected para asegurar la inyeccion de información, y que el provideIn tenga la ‘root’ declarada (osea la bd que inyecta la info)



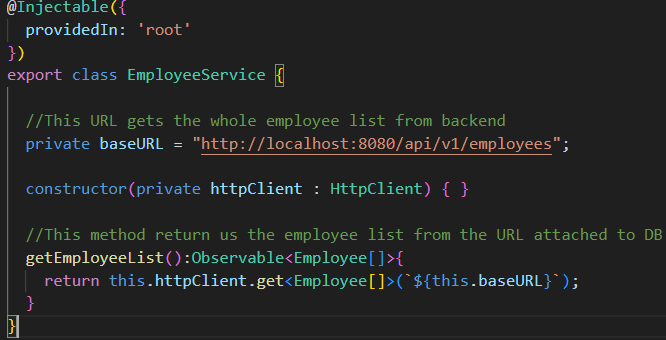
Ahora en el archive app.module.ts vamos a importar “HttpClientModule” y asegurarnos de escribir manualmente el import en la cabecera



15 – Vamos a employee.service.ts y declaramos dentro de la clase un atributo privado “baseURL” que contiene la direccion del getList del backend (CLAVE: el this.baseURL tiene que estar encerrado por ``, no pueden ser ‘’ ni ningún otro caracter…)

Luego en el constructor ponemos argumentos, “(private httpClient : HttpClient)” e importamos.

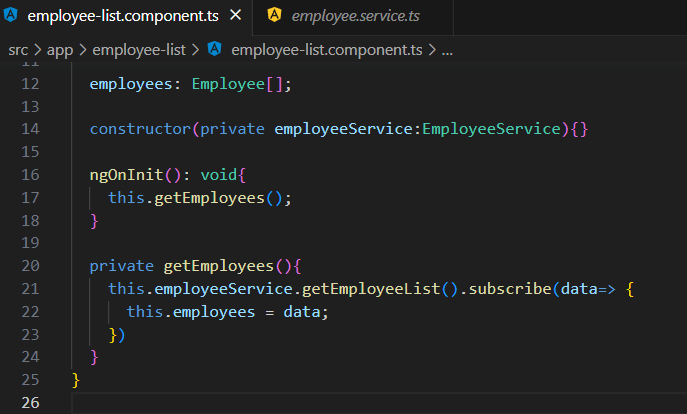
Luego creamos un metodo para retornar el arreglo de empleados



16 – Volvemos a employee.list.components.ts y al constructor de la clase le enviamos como argumentos “private employeeService:EmployeeService”

Luego creamos un metodo llamado private getEmployees() y hacemos que subscriba al obersvable de employeeService

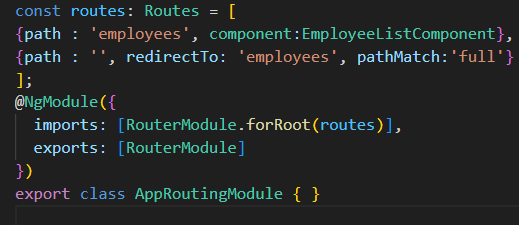
Modificamos el ngOnInit() borrando la info hardcodeada y en su lugar llamado al getEmployees()



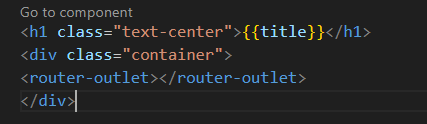
En Eclipse, al archive EmployeeController le agregamos la anotación @CrossOrigin(origins=””) e importamos.  
Esto nos va a permitir cruzar la información entre el backend y el frontend

17 – En app-routing.module.ts vamos a cofigurar las rutas que vamos a utilizar cuando entramos con navegador

En las rutas agregamos las siguientes lineas para redireccionar los links a la lista de empleados

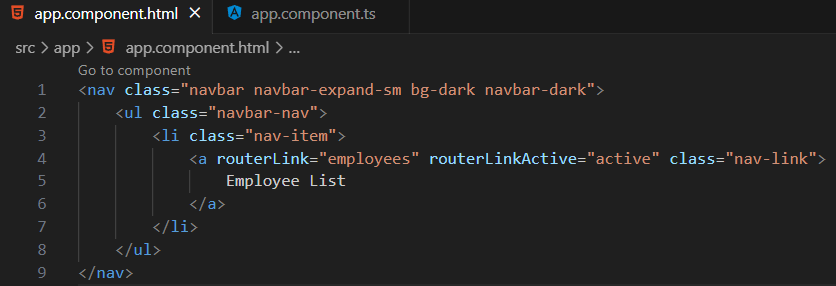


18 – en app.component.html cambiamos la línea de app-component-list por router-outlet



En app.component.ts cambiamos el title por el que querramos. Hacemos lo mismos en index.html

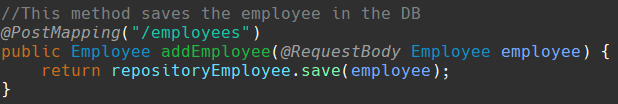
Vamos a app.component.html y previo al h1 codeamos lo siguiente



Con este Código vamos a crear una barra de navegación al comienzo de la pantalla con un tono obscuro

19 – Volvemos al backend y en el controller creamos el metodo addEmployee. Agregamos la anotación de @PostMapping y en los args le agregamos la anotación de @BodyRequest Employee employee para que sepa que por parametros debe recibir al empleados que va a grabar.

Luego, en el return lo grabamos mediant el metodo save



20 – En VS creamos una nueva terminal y vamos a crear un nuevo componente llamado add-employee

Tipeamos “ng g c add-employee” G de generate y c de component

En app-routing.models.ts agregamos el path 

Vamos a app.component.html y agregamos al nav el link para ir a add employee

 <li class="nav-item">

        <a routerLink="add-employee" routerLinkActive="active" class="nav-link">

                Add employee</a></li>

21 – En app.module.ts en la parte de las importaciones, agregamos FormModule

imports: [

    BrowserModule,

    AppRoutingModule,

    HttpClientModule,

    FormsModule

  ]

En add-employee.components.ts dentro de la clase creamos un nuevo empleado y un metodo llamado onSubmit() donde mostraremos por console al empleado

export class AddEmployeeComponent {

  employee: Employee = new Employee();

  constructor() { };

  ngOnInit(): void {

}

onSubmit(){

    console.log(this.employee);

  }

Ahora, en add-employee.component.html vamos a crear el form para la carga de los datos

Utilizando el ngModel en el input, vamos a cruzar los datos que se reciben desde el onSubmit() y crear el nuevo empleado para agregar de la clasee AddEmployeeComponent



Hacemos 3 divs, uno por cada atributo (name, surname, email; todo en minuscula)

(tener en cuenta que es desde el div interno y que en el div del Email el type es “email”

Por ultimo, agregamos un botón “add” para que se envién los datos

<button class="btn btn-succes"> Add </button>

En employee.service.ts voy a crear el metodo addEmployee que recibe un employee y retorna un oberservable formado por la conjunction de la url y el empleado

 addEmployee(employee:Employee):Observable<Object>{

    return this.httpClient.post(`${this.baseURL}`, Employee);

  }

En add-employee.component.ts vamos a modificar el constructor y vamos a enviarle mediante los argumentos un EmployeeService y un Router

constructor(private employeeService: EmployeeService, private router: Router) { };

Luego creamos el metodo addEmployee que no recibe nada y lo que hace es subscribirse al obersvable que retorna el metodo anterior, luego tiene una function arrow por la que envía la info recibida (el empleado)

Por ultimo agregamos un error y un console log que muestra el error

 addEmployee() {

    this.employeeService.addEmployee(this.employee).subscribe(data => {

      console.log(data);

this.backToList();

    }, error => console.log(error));

  }

Creamos un método para que nos re-direccione a la list de empleados post add

backToList(){

    this.router.navigate(['/employees']);

  }

Agregamos el metodo al addEmployee

Ahora modificamos el onSubmit (lo que hará la app una vez que apretes el boton) para que llame al metodo add

 onSubmit() {

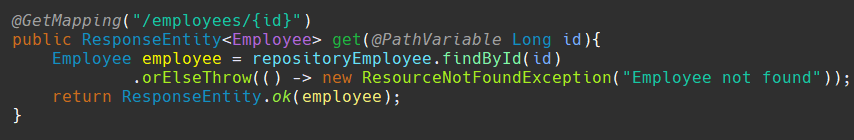
    this.addEmployee();

  }

Con esto, el método add ya debería agregar el employee nuevo a la DB

22- Ahora vamos a crear el método para buscar un empleado

En Eclipse en la carpeta de Controller creamos el siguiente metodo

Retorna una entidad que va a ser empleado, recibe por parametros el ID a buscar

Dentro del metodo utiliza el findById(idBuscado) y en caso de que no lo recupere tira el error que habiamos personalizado

23- Vamos a generar el método para realizar modificaciones.   
En el controller de Java creamos el metodo update que va a recibir (@PathVariable Long id, @RequestBody Employee modifiedEmployee)   
Al igual que en “get” primero tenemos que utilizar el id que recibimos por parametro para buscar al empleado, en caso de que no exista tira un error.   
Una vez encontrado el empleado le seteamos los nuevos valores que vienen por el bodyRequest  
Por ultimo instanceamos un nuevo empleado y lo grabamos mediante el metodo SAVE de repositoryEmployee y lo devolvemos en el return

*@PutMapping*("/employees/{id}")

public ResponseEntity<Employee> update(*@PathVariable* Long id, *@RequestBody* Employee modifiedEmployee){

Employee employee = repositoryEmployee.findById(id)

.orElseThrow(() -> new ResourceNotFoundException("Employee not found"));

employee.setName(modifiedEmployee.getName());

employee.setSurname(modifiedEmployee.getSurname());

employee.setEmail(modifiedEmployee.getEmail());

Employee updatedEmployee = repositoryEmployee.save(employee);

return ResponseEntity.*ok*(updatedEmployee);

}

Ahora vamos a crear en el front la vista del update. En la consola escribimos “ng g c employee-update” para crear el componente del update

Luego en “employee-list.component.html” debajo del <th>Email<th> agregamos un th de Actions (el th es para crear una columna donde apareceran las acciones posibles… editar, eliminar y otras) .  
Agregamos debajo de las celdas un boton para editar el empleado

    <tbody>

        <tr \*ngFor="let employee of employees">

            <td>{{employee.name}}</td>

            <td>{{employee.surname}}</td>

            <td>{{employee.email}}</td>

            <td>

 <button (click)="updateEmployee(employee.id)" class="btn btn-info">Edit</button>

            </td>

        </tr>

    </tbody>

</table>

En el archive “employee-list.component.ts” agregamos al constructor un router del tipo Router privado.  
Creamos el metodo updateEmployee que recibe un id del tipo number y utilizamos el router para navegar y recuperar el id

updateEmployee(id:number){

    this.router.navigate(['employee-update', id]);

  }

Vamos a el archive app-routing.module.ts y agregamos el path (osea la url a la que nos direccionara cuando apretemos el boton de edit)

{path : 'update-employee/:id', component : UpdateEmployeeComponent}

Y en la parte de import ponemos esto si es que no aparece automaticamente

import { UpdateEmployeeComponent } from './employee-update/employee-update.component';

Ahora en employee.service.ts vamos a crear el metodo getEmployee que nos va a devolver a un empleado especifico que busquemos por id, esto nos va a ayudar a recuperar elk empleado al que queremos editar

  //This method retrieves a certain employee from DB using his id

  getEmployee(id:number): Observable<Employee>{

    return this.httpClient.get<Employee>(`${this.baseURL}/${id}`);

  }

El metodo recibe un id y devuelve un observable del tipo Employee. En el return está la url en la que si ponemos el id encontraremos a dicho empleado

Una vez terminado esto, creamos el metodo para hacer el update

updateEmployee(id:number, employee:Employee): Observable<Object>{

    return this.httpClient.put(`${this.baseURL}/${id}`, employee);

  }

Recibe un id (el que luego se usara en el metodo getEmployee) y un Employee, retorna un observable y en el return devuelve la direccion url que nos da un x empleado y un empleado

Ahora vamos a trabajar en update-employee.component.ts :

Primero instalamos la api ‘sweetalert’ con el commando npm install sweetalert2

Esto lo escribimos en la terminal

Luego importamos “swal” de dicha api

import swal from 'sweetalert2';

Ahora para trabajar en la clase en si le agregamos que implementa OnInit.

Le agregamos un atributo del tipo empleado y un id del tipo number.

Hacemos un constructor con los siguientes parametros

constructor(private employeeService: EmployeeService, private router: Router, private route:ActivatedRoute) { };

creamos el metodo ngOnInit() que retorna void

ngOnInit(): void {

    this.id = this.route.snapshot.params['id'];

    this.employeeService.getEmployee(this.id).subscribe(data => {

      this.employee = data;

    }, error => console.log(error))

  }

El route.snapshop va a contener el argumento que viene por parametron, en este caso es el id

UItilizamos el metodo getEmployee para recuperar el empleado que queremos modificar y subscribimos al observable de dicho metodo, la info del empleado sera recuperada por la function arrow en la variable data y luego this.employee lo va a guardar

Si hay error, lo mostramos por consola

Creamos un metodo llamado backtoList() para que luego de editado el empleado nos devuelva a la lista de empleados automaticamente

Utilizamos el router para navegar nuevamente a la url de la lista de empleados y luego utilizamos la api para que cuando el empelado sea actualizado con exito nos muestre un cartel interactivo

Dicho metodo (swal.fire) comprende 3 partes. La primera sera el mensaje que figura como titulo en el cartel, el Segundo es el mensaje explayado de lo sucedido en el que agregamos el nombre del empleado para que figure y lo ultima es el tipo de suceso.

 backToList() {

    this.router.navigate(['/employees']);

    swal.fire('Updated employee', `The employee ${this.employee.name} has been succesfuly updated`, `success`);

  }

Por ultimo creamos el onSubmit, que marca lo que sucede cuando se da click en el boton aceptar

Este llama al metodo updateemployee y subscribe al resultado del observable que habiamos creado previamente en employee.service.ts

Este metodo llama al backToList() y si hay un error lo muestra por consola

onSubmit() {

    this.employeeService.updateEmployee(this.id, this.employee).subscribe(data => {

      this.backToList();

    }, error => console.log(error))

  }

Ahora viene la parte de crear el component html de update-employee, no voy a explicarlo, pero se hace uso de Angular para corroborar que los datos tengan los parametros deseados (min or max length) y que sean del tipo necesitado

<div class="col-md-6 offset-md-3">

    <h3 class="text-center">Update an employee</h3>

    <form (ngSubmit)= "onSubmit()" #employeeForm = "ngForm">

      <div class="form-group">

        <label>Name</label>

        <input type = "text"

        class="form-control"

        id = "name"

        [(ngModel)] = "employee.name"

        name = "name"

        required

        minlength="4"

        maxlength="10"

        placeholder="Type a name"

        #name = "ngModel"

        >

        <div class="alert alert-danger" \*ngIf= "name.invalid && (name.dirty || name.touched)">

          <div \*ngIf = "name.errors?.['required']">

            Type a name

          </div>

          <div \*ngIf = "name.errors?.['minlength']">

            Name must contain at least 4 characters

          </div>

          <div \*ngIf = "name.errors?.['maxlength']">

            Name must containg a maximum of 10 characters

          </div>

        </div>

      </div>

      <div class="form-group">

        <label>Surname</label>

        <input type = "text"

        class="form-control"

        id = "surname"

        [(ngModel)] = "employee.surname"

        name = "surname"

        required

        minlength="4"

        maxlength="10"

        placeholder="Type a surname "

        #surname = "ngModel"

        >

        <div class="alert alert-danger" \*ngIf= "surname.invalid && (surname.dirty || surname.touched)">

          <div \*ngIf = "surname.errors?.['required']">

            Type a surname

          </div>

          <div \*ngIf = "surname.errors?.['minlength']">

            Surname must contain at least 4 characters

          </div>

          <div \*ngIf = "surname.errors?.['maxlength']">

            Surname must containg a maximum of 10 characters

          </div>

        </div>

      </div>

      <div class="form-group">

        <label>Email : </label>

        <input type = "email"

        class="form-control"

        id = "email"

        [(ngModel)] = "employee.email"

        name = "email"

        pattern ="[a-zA-Z0-9\_]+([.][a-zA-Z0-9\_]+)\*@[a-zA-Z0-9\_]+([.][a-zA-Z0-9\_]+)\*[.][a-zA-Z]{1,5}"

        required

        placeholder="Digite su email "

        #email = "ngModel"

        >

        <div class="alert alert-danger" \*ngIf= "email.invalid && (email.dirty || email.touched)">

          <div \*ngIf = "email.errors?.['required']">

            Type an email

          </div>

          <div \*ngIf = "email.errors?.['pattern']">

            Type an email with the following format "something@something.com"

          </div>

        </div>

      </div>

      <button class="btn btn-success" [disabled]= "!employeeForm.form.valid">Save</button>

    </form>

  </div>

24 – Vamos a crear el metodo deleteEmployee ahora

En el backend creamos el metodo delete que utiliza el metodo findById(id) de respositoryEmployee para buscar el empleado que quiere eliminar, en caso de que no exista, tira error (el que creamos)

//This method deletes a specific employee

*@DeleteMapping*("/employees/{id}")

public ResponseEntity<Map<String,Boolean>> deleteEmployee(*@PathVariable* Long id){

Employee employee = repositoryEmployee.findById(id)

.orElseThrow(() -> new ResourceNotFoundException("Employee doesn't existe"));

repositoryEmployee.delete(employee);

Map<String,Boolean> answer = new HashMap<>();

answer.put("delete", Boolean.***TRUE***);

return ResponseEntity.*ok*(answer);

}

Ahora en el front en el html de employee-list agregamos el boton delete

<button (click)="deleteEmployee(employee.id)" class="btn btn-danger" style="margin-left: 10px;">Delete</button>

Ahora vamos a crear el metodo delete

Vamos a employee.service.ts y agregamos el metodo delete que recibe por parametron el id a eliminar y retorna un observcable, en el return del metodo tenemos la id con un .delete y la url del empleado buscado

 deleteEmployee(id:number): Observable<Object>{

    return this.httpClient.delete(`${this.baseURL}/${id}`);

  }

Ahora en employee-list.component.ts creamos el metodo deleteEmployee que recibe por params el id del empleado a borrar… este metodo se subscribe al observable de arriba

Por console log manda los datos recuperados y llama a this.getEmployees()

 deleteEmployee(id:number){

    this.employeeService.deleteEmployee(id).subscribe(data => {

      console.log(data);

this.getEmployees();

  })

}

(este es el metodo al que llama el boton delete de employee-list.html, luego este metodo llama al de service.ts quien es el que ejecuta realmente la accion)

25 – Creamos ahora el componente para ver los detalles del empleado

En la terminal escribimos ng g c employee-details

En el app routing agregamos el path del employee details

{path : 'employee-details/:id', component : EmployeeDetailsComponent}

import { EmployeeDetailsComponent } from './employee-details/employee-details.component';

Ahora vamos al employee-list.component.html para agregar el boton de detalles

<button (click)="seeDetails(employee.id)" class="btn btn-primary" style="margin-left: 10px;">Details</button>

Ahora en employee-list.component.ts creamos el metodo seeDetails()

 seeDetails(id:number){

    this.router.navigate(['employee-details', id]);

  }

Recibe un id por param y navega hacia la url deseada

Vamos a employee-details.component.ts y le agregamos un atributo id:number y un employee:Employee

Al contructor le agregamos una route:ActivatedRoute y un employeeService:EmployeeService, ambos privados

Luego en el ngOnInit():void al atributo this.id le agregamos el id que viene por parametron mediant eel snaposhot e instanceamos un empleado.

Utilizamos el employeeService para llamar y subscribirnos al resultado de getEmployee(this.id), este lo capturamos en el this.employee

export class EmployeeDetailsComponent {

  id:number;

  employee:Employee;

  constructor(private route:ActivatedRoute, private employeeService:EmployeeService){}

  ngOnInit(): void{

    this.id = this.route.snapshot.params['id'];

    this.employee = new Employee();

    this.employeeService.getEmployee(this.id).subscribe(data =>{

      this.employee = data;

    })

  }

}

Ahora vamos al html de employee details y creamos un H3 con el mensaje employee’s details

Luego un div dentro del cual tendrá varios div con el label de cada atributo del empleado y luego su respective valor

<h3>Employee's details</h3>

<div>

    <div>

        <label><b>ID: </b>{{employee.id}}</label>

    </div>

    <div>

        <label><b>Name: </b>{{employee.name}}</label>

    </div>

    <div>

        <label><b>Surname: </b>{{employee.surname}}</label>

    </div>

    <div>

        <label><b>Email: </b>{{employee.email}}</label>

    </div>

</div>

Con esto terminamos el CRUD básico

Ahora vamos a la parte de validar los datos

26 – Primero vamos al tsconfig.json y modificamos la linea “strict” de true a false para que no permita nulleables

"strict": false,

Y en la parte de AngularCompilerOptions agregamos las siguientes lienas con la misma finalidad

    "strictFuntionTypes": false,

    "strictBindCallApply": false,

    "strictPropertyInitialization": false

Vamos a add-employee.component.html y le agregamos la propiedad de requerido.

Le ponemos un min y un max length, un valor al placeholder “insert your name”, le agregamos el #name = “ngModel” (esto recupera el atributo name del empleado de la clase y lo mismo hara con los otros atributos) y creamos un div donde preguntaremos si el nombre que tiene el empleado es valido, en caso de que no se lanzara un mensaje con el valor a modificar

required

            minlength="4"

            maxlength="10"

            placeholder="Insert your name"

            #name= "ngModel">

            <div class="alert alert-danger" \*ngIf="name.invalid && (name.dirty || name.touched)">

                <div \*ngIf = "name.errors?.['required']">

                    Write a name

                </div>

                <div \*ngIf = "name.errors?.['minlength']">

                    Name must have 4 characters minimum

                </div>

                <div \*ngIf = "name.errors?.['maxlength']">

                    Name must have 10 characters maximum

                </div>

            </div>

Para validar el surname es exactamente igual cambiando los valores a controlar

Para email la diferencia es que hay que agregar un pattern para hacer la validacion de los caracteres del email y utilizar el ngIf para controlar dicho error

name = "email"

            pattern="[a-zA-Z0-9.!#$%&'\*+/=?^\_`{|}~-]+@[a-zA-Z0-9-]+(?:\.[a-zA-Z0-9-]+)\*$"

            required

            placeholder="Insert your email"

            #email= "ngModel">

            <div class="alert alert-danger" \*ngIf="email.invalid && (email.dirty || email.touched)">

                <div \*ngIf = "email.errors?.['required']">

                    Write an email

                </div>

                <div \*ngIf = "surname.errors?.['pattern']">

                    Wrong pattern (example something@else.com)

                </div>

            </div>

Ahora volvemos al top y en el form le agregamos el atributo #employeeForm

<form (ngSubmit)="onSubmit()" #employeeForm = "ngForm">

Por ultimo vamos al boton y lo deshabilitamos en caso de que el form no este complete

<button class="btn btn-succes" [disabled]="!employeeForm.form.valid"> Add </button>

(notese que se pregunta si es valido y se anula con el !)

Ahora copiamos todo el codigo del html y lo pegamos tal cual en el employee update.component.html para que corra las mismas limitaciones

27 – Por ultimo modificamos el metodo delete de employee-list para que utilice la api de sweetalert

  deleteEmployee(id:number){

    Swal.fire({

      title : 'Are you sure?',

      text : 'Confirm if you want to delete this employee',

      showCancelButton: true,

      confirmButtonColor: '#3085d6',

      cancelButtonColor: '#d33',

      confirmButtonText: "Confirm",

      cancelButtonText: "Cancel",

      buttonsStyling: true

    }).then((result) => {

      if(result.value){

        this.employeeService.deleteEmployee(id).subscribe(data => {

          console.log(data);

          this.getEmployees();

          swal.fire('Employee deleted', "Employee has been deleted", 'success')

        })

      }

    })

  }

Esto genera un cuadro que pregunta si se quiere borrar el empleado, en caso de que si muestra un cartel, caso contrario no hace nada

FIN