*Àlex Macià Fiteni*​

Práctica 3 Opcional – Login y Rutas

Añadiendo login de usuario a la aplicación ToDo

NodeJS, Angular, API RESTFul

# Objetivo de la práctica

El objetivo de esta práctica es extender la funcionalidad de nuestra **ToDo App** desarrollada en la práctica 3 para que soporte un sistema de usuarios. En esta segunda versión, se podrá registrarse en la página y luego iniciar sesión, de manera que cada usuario verá sólo sus tareas.

Para conseguirlo habrá que modificar el servidor API RESTFul, pero sobre todo la propia aplicación ToDo.

Si quiere saber más sobre el desarrollo de la aplicación distribuida y todos sus detalles véase la memoria de la práctica 3.

# Herramientas

Utilizaremos las mismas herramientas que utilizamos en el desarrollo de ToDo App:

* Un **editor de código:** El elegido en este caso es VSCode, pues conoce NodeJS de manera nativa y es más fácil leer el código.
* Un **navegador convencional** para ver la aplicación. Hemos elegido Firefox.
* Un **cliente web** más versátil que un navegador, capaz de realizar las pruebas que hagan falta. Al igual que en la práctica anterior utilizaremos Postman

# Desarrollo de las funcionalidades

El primer cambio que vamos a realizar es en el **API RESTful** para que nos permita iniciar sesión. Lo único que necesitamos añadir son nuevas rutas específicas para el inicio de sesión y la recuperación de tareas de un usuario específico. La creación de usuarios será una simple entrada de datos en una colección **users**, así que el **POST** que ya tenemos nos vale.

El inicio de sesión es con **POST** y enviando el nombre de usuario y la contraseña en el **body** al recurso **api/sessions**. Como tenemos una ruta genérica para cualquier **api/:coleccion** tendremos que poner esta nueva ruta por encima del resto de **POSTs** para que salte antes (de otra manera, **api/sessions** sería interpretado como **api/:coleccion**, siendo colección = sessions). Devolverá el objeto del usuario de la base de datos, y en la aplicación de Angular recuperaremos su ID. Sería algo así:

**Fragmento 1.1:** Ruta para inicio de sesión

app.post('/api/sessions', (req, res, next) => {

const userData = req.body;

console.log(userData);

req.collection = db.collection("users");

req.collection.findOne({username: userData.username, password: userData.password}, (err, elemento) => {

if(err) return next(err);

res.json(elemento);

});

});

Para recuperar tareas de un usuario específico necesitamos otra excepción. Antes de nada tengamos en cuenta que las tareas ahora van a ser creadas con un campo **user** que indicará la ID de Mongo del usuario propietario. Esta vez queremos un **GET** al recurso **api/tasks/:userID**, en la que userID es la id del usuario a filtrar en la colección **tasks**. De nuevo hay que añadirla sobre otros GET para que las llamadas más genéricas no intervengan en su lugar. Sería algo así:

**Fragmento 1.2:** Tareas de un solo usuario

app.get('/api/tasks/:user', (req, res, next) => {

req.collection = db.collection("tasks");

req.collection.find({user: req.params.user}).toArray((err, coleccion) => {

if(err) return next(err);

res.json(coleccion);

});

});

Es muy importante que estas cosas se calculen aquí, en el lado del servidor. Si para mostrar las tareas de un usuario las bajáramos todas en la app y las filtráramos ahí, habríamos cargado las tareas de todo el mundo, lo que supone una falta de confidencialidad (cualquiera podría ver las tareas de los demás) a parte de un malgasto de memoria del cliente. Con todo, el archivo final queda así:

**Archivo 1:** index.js

'use strict'

// Asigna el puerto a la variable de entorno. Si no existe coge el 3000

const port = process.env.PORT || 3000;

const express = require('express');

const logger = require('morgan');

const mongojs = require('mongojs');

const cors = require('cors');

const app = express();

var db = mongojs("SD");

var id = mongojs.ObjectID;

var allowCrossTokenHeaders = (req, res, next) => {

res.header("Access-Control-Allow-Headers", "\*");

return next();

};

var allowCrossTokenOrigin = (req, res, next) => {

res.header("Access-Control-Allow-Origin", "\*");

return next();

};

// Middlewares

app.use(logger('dev'));

app.use(express.urlencoded({extended:false}));

app.use(express.json());

app.use(cors());

app.use(allowCrossTokenHeaders);

app.use(allowCrossTokenOrigin);

// Anyadimos un trigger para cambiar de coleccion dinamicamente

app.param("coleccion", (req, res, next, coleccion) => {

req.collection = db.collection(coleccion);

return next();

});

// Routes

app.get('/api', (req, res, next) => {

db.getCollectionNames((err, colecciones) => {

if(err) return next(err);

res.json(colecciones);

});

});

app.get('/api/tasks/:user', (req, res, next) => {

req.collection = db.collection("tasks");

req.collection.find({user: req.params.user}).toArray((err, coleccion) => {

if(err) return next(err);

res.json(coleccion);

});

});

app.get('/api/:coleccion', (req, res, next) => {

req.collection.find((err, coleccion) => {

if(err) return next(err);

res.json(coleccion);

});

});

app.get('/api/:coleccion/:id', (req, res, next) => {

req.collection.findOne({\_id: id(req.params.id)}, (err, elemento) => {

if(err) return next(err);

res.json(elemento);

});

});

app.post('/api/sessions', (req, res, next) => {

const userData = req.body;

console.log(userData);

req.collection = db.collection("users");

req.collection.findOne({username: userData.username, password: userData.password}, (err, elemento) => {

if(err) return next(err);

res.json(elemento);

});

});

app.post('/api/:coleccion', (req, res, next) => {

const elemento = req.body;

// Save es capaz de crear o actualizar una entrada. Es como si

// automaticamente hiciese insert o update en funcion de lo que necesite

req.collection.save(elemento, (err, elementoGuardado) => {

if(err) return next(err);

res.json(elementoGuardado);

});

});

app.put('/api/:coleccion/:id', (req, res, next) => {

const elemento = req.body;

console.log(elemento);

// update nos pide una query json para actualizar la entrada especificada

// En nuestro caso esta query sera la id que nos digan en los parametros

// Tambien pasamos los nuevos datos

req.collection.update({\_id: id(req.params.id)}, elemento, (err, resultado) => {

if(err) return next(err);

res.json(resultado);

});

});

app.delete('/api/:coleccion/:id', (req, res, next) => {

// remove nos pide una query json para borrar la entrada especificada

// En nuestro caso esta query sera la id que nos digan en los parametros

req.collection.remove({\_id: id(req.params.id)}, (err, resultado) => {

if(err) return next(err);

res.json(resultado);

});

});

// Lanzamos nuestro servicio

app.listen(port, ()=>{

console.log(`API REST ejecutándose en http://localhost:${port}/api`);

});

Ahora podemos desarrollar la funcionalidad de la app en **Angular**. En la aplicación vamos a añadir varias cosas nuevas: una clase **usuario**, un componente **login**, un comoponente **registro**, un servicio de **usuarios** (para registrarse e iniciar sesión) y los cambios pertinentes en **app.component**.

Vamos a empezar por la clase usuario. Nos dirigimos a la carpeta **src/app** y creamos el archivo **User.ts**.

Luego escribimos lo siguiente:

**Archivo 2:** User.ts

// Creamos la clase User

export class User {

\_id?: string;

username: string;

password: string;

email?: string;

}

Ahora vamos a crear el servicio users que realice las llamadas al API. Vamos desde la terminal al directorio raíz de la aplicación y ejecutamos:

**$ ng g s services/user**

Ahora vamos en el editor de código al archivo **user.service.ts** en **src/app/services/** y nos disponemos a terminarlo. El procedimiento es similar al que seguimos para crear el servicio task: Primero importamos el módulo de cliente Http, el de map, y esta vez la clase usuario

**Fragmento 3.1:** Imports

import { HttpClient } from '@angular/common/http';

import { map } from 'rxjs/operators';

import { User } from '../User';

Ahora en la clase **UserService** declaramos la propiedad **apiREST** que contiene la url al api:

**Fragmento 3.2:** Ruta al API REST

apiREST = "http://localhost:3000/api";

Luego en el constructor declaramos un nuevo **HttpClient**.

**Fragmento 3.3:** Constructor

constructor(private http: HttpClient) { }

Por último declaramos los métodos para registrarse y para iniciar sesión:

**Fragmento 3.4:** Métodos

userRegister(newUser: User) {

return this.http.post<User>(`${this.apiREST}/users`, newUser)

.pipe(map(res=>res));

}

userLogin(userData: User) {

return this.http.post<User>(`${this.apiREST}/sessions`, userData)

.pipe(map(res=>res));

}

Fíjate que hemos usado para el login un **POST** a la ruta que hemos creado antes de **api/sessions**, pero para registrar al usuario usamos un **POST** a la colección **users** con los datos del usuario.

Con todo, el archivo queda así:

**Archivo 3:** user.service.ts

import { Injectable } from '@angular/core';

import { HttpClient } from '@angular/common/http';

import { map } from 'rxjs/operators';

import { User } from '../User';

@Injectable({

providedIn: 'root'

})

export class UserService {

apiREST = "http://localhost:3000/api";

constructor(private http: HttpClient) { }

userRegister(newUser: User) {

return this.http.post<User>(`${this.apiREST}/users`, newUser)

.pipe(map(res=>res));

}

userLogin(userData: User) {

return this.http.post<User>(`${this.apiREST}/sessions`, userData)

.pipe(map(res=>res));

}

}

Ahora procederíamos a desarrollar los componentes login y registro, pero antes vamos a configurar las **rutas** para luego poder cambiar entre los componentes login, registro y tareas. Recordemos que las apps de **Angular** son de una sola página que carga distintos componentes.

Vamos **app.module.ts** y añadimos los import

import { RouterModule, Routes } from '@angular/router';

Luego creamos un array de rutas. Las rutas las define como objetos **JSON** con un formato específico

const appRoutes: Routes = [

{path: 'tasks', component: TasksComponent},

{path: 'login', component: LoginComponent},

{path: 'register', component: RegisterComponent},

{path: '', redirectTo: '/tasks', pathMatch: 'full'}

];

Todas las rutas tienen el parámetro **path** que indica la ruta que definen. Luego la mayoría tienen un **componente** asignado que es el que cargan si su ruta coincide. La última ruta, la ruta vacía, es la que se activa si no hay recurso en la URL. En este caso le hemos dicho que nos redirija a la ruta tasks. Además tiene el parámetro **pathMach** a full, ya que por defecto podría activarse por que la ruta en la url contenga parcialmente esa ruta, y la ruta vacía es una subcadena de cualquier ruta. En cualquier caso no se activaría, pues el orden importa y el router, cuando carguemos una ruta, buscará la primera que coincida en esta lista en orden.

Por último añadimos el módulo al array import:

RouterModule.forRoot(appRoutes, { enableTracing: true })

El parámetro **enableTracing** está por cuestiones de **debug**. Te va a llenar la consola de datos cada vez que algún evento del **router** se dispare. Ahora sí podemos desarrollar los componentes login y registro.

Desde la terminal, en el directorio raíz de la aplicación, ejecutamos

**$ ng g c component/login**

**$ ng g c component/register**

Ahora vamos al archivo **login.component.ts** en **src/app/component/login/** y lo abrimos con el editor de código.

app.component.ts → login.component.ts → login.component.html → register.component.ts → register.component.html → Task.ts → task.service.ts → app.component.html

# Pruebas de funcionamiento

El ob

# Bibliografía

Las referencias que se han utilizado son las siguientes:

**Rutas en Angular**

<https://angular.io/guide/router>

**Uso de Angular**

<https://angular.io/api/common/NgIf>

**Querys de find a mongo en Node**

<https://flaviocopes.com/node-mongodb/>