26-9-2025

DESPLEGANDO NUESTRA PROPIA APLICACIÓN

DAW2 - DAW

DESPLEGANDO NUESTRA PROPIA APLICACIÓN

EXPLICANDO MI PROYECTO

En esta actividad, se nos indica que debemos de desplegar o hacer funcionar una aplicación completa. También se nos insta a decidir lo que queremos hacer. Podemos elegir entre usar lo visto de servlets hasta el momento o si que remos aportar nuestro grano de arena añadiendo o implementando alguna tecnología distinta.

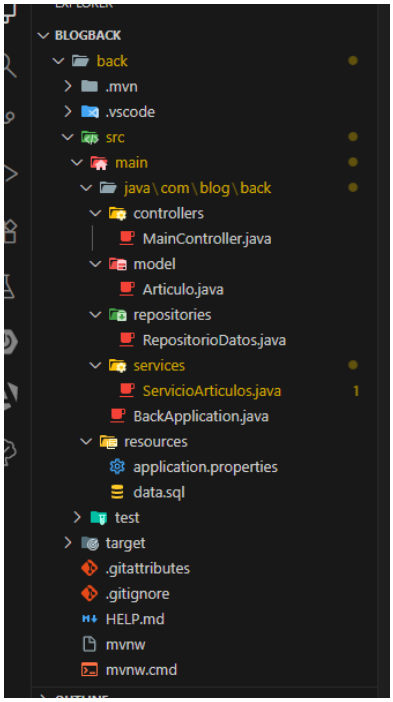
Yo para este caso he optado por calentarme, como suele ser habitual, con lo que ello conlleva, como por ejemplo que las cosas no salgan o se tuerzan o que esté hasta el último minuto intentando que funcione porque no llego. En este caso, todas a la vez. Decidí juntar lo que estamos viendo en el front, Angular V20, y lo que estamos viendo en el back, SpringBoot. Además, viendo alguna tecnología, decidí darle a la aplicación una falsa "persistencia". Esto quiere decir que use una dependencia de SprinBoot, JPA para bases de datos, incluyendo una base de datos H2, que es muy ligera, no requiere de crear un servidor de BD adicional y, en mi caso, la configuré para que se almacenase en memoria, o lo que es lo mismo, que se borrase cuando la aplicación o el servidor se detuviese. Para mejorar un poco esto busqué configuraciones que me permitiesen cargar en el inicio una serie de datos, para poder hacer pruebas y que se mostrase algo al inicio.

Para aprender Angular, que todavía estaba y estoy muy verde, usé de base un proyecto de esa asignatura, hacer un blog. Me pareció que tenía las opciones que necesitaba para mostrar algo funcional. En un principio iba a ser un To-Do-List sencillo, pero ya que empecé haciendo el blog, continué por ese camino, ya que en esencia iba a hacer las mismas operaciones.

Ahora ya, comienzo explicando en que consiste mi aplicación.

EL BACK-END EN UNA API DE SPRINGBOOT

Como ya he explicado antes decidí hacer el back usando SpringBoot, así que lo primero que hice fue crear un proyecto en VS Code usando Spring Initializer, para darle la configuración inicial.



Como vemos en la imagen, se crea algo más o menos como esto, con menos elementos, pero todos nos hacemos una idea de como es. Durante el proyecto tuve que añadir las carpetas de controllers, model, repositories, services y añadir las clases que vemos en la imagen. Además en la carpeta resources, vemos un elemento, data.sql del que hablaré después.

ILUSTRACIÓN 1 ESQUELETO DEL BACK

Ahora voy a explicar paso a paso brevemente lo que es cada archivo que vemos en la imagen, Menos el BackApplication.java, que es el lanzador por defecto, sin ninguna modificación.

MAIN CONTROLLER

Este es el archivo en el que voy a definir mis endpoints, las direcciones a las que se conectará el Front para trabajar con datos.

```
@RestController
@CrossOrigin
@RequestMapping("/api")

public class MainController {


    @Autowired
    private ServicioArticulos servicioArticulos;

    @GetMapping("/articulos")
    public List<Articulo> obtenerArticulos() {
        return servicioArticulos.getArticulos();
    }

    @PostMapping("/save")
    public void postMethodName(@RequestBody Articulo articulo) {

        servicioArticulos.guardarArticulo(articulo);

    }


}
```

ILUSTRACIÓN 2 MAIN CONTROLLER

Este archivo es muy sencillo. Me hubiese gustado tener más tiempo para añadir alguna operación más, pero dio lo que dio. Lo explico brevemente.

Las anotaciones superiores (@RestController, @CrossOrigin y @RequestMapping "/api") las necesito para indicar al controller como funciona. La primera le indica que el front está en un dominio o ip diferente al back, la segunda para anular las cabeceras CORS de las peticiones HTTP, que me han dado muchos problemas en otros proyectos. La última indica el punto de entrada raíz a la aplicación que se debe de incluir en la barra de direcciones web.

Luego vemos un atributo @Autowired. Es una instancia de la clase servicios y Autowired hace que no me tenga que preocupar de inicializar una

instancia de ese objeto, SpringBoot buscará una clase indicada como @Service que coincida con el nombre que yo le indico y manejará su definición.

Por último tenemos dos métodos, @GetMapping y @PostMapping. Uno se encargará de suministrar todos los datos al front, el Get, y el Post se encargará de almacenar los datos en la base de datos. Estos dos métodos se apoyan en la clase Servicio que he definido para ejecutar sus tareas.

EL MODEL

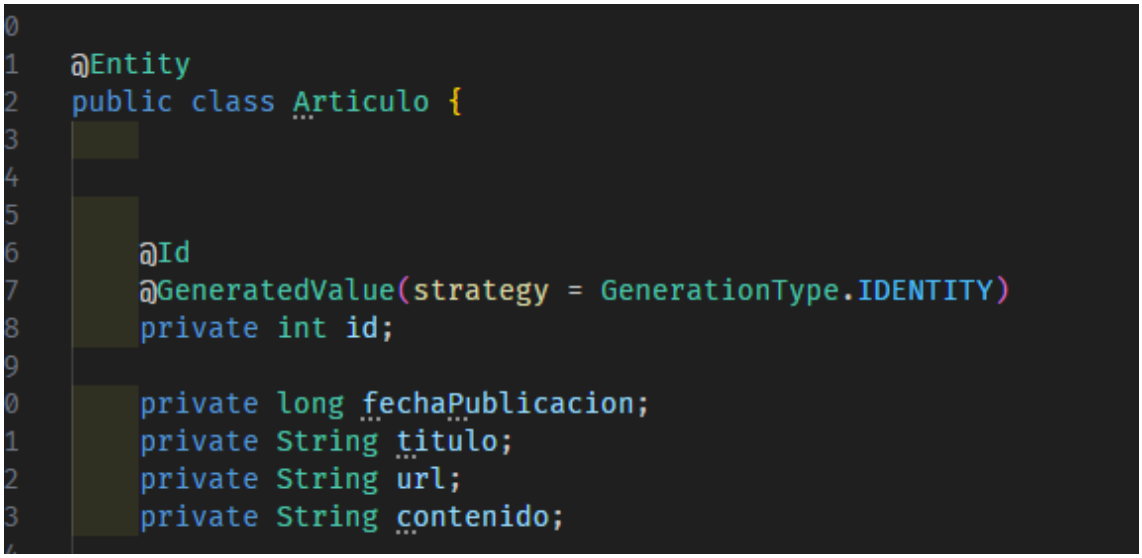


ILUSTRACIÓN 3 LA CLASE ARTICULO

En esta clase defino un objeto de la forma clásica que se hace en Java, pero con las especificaciones de SpringBoot.

Primero añado la notación @Entity, que me ayudará a que el módulo JPA trate a estos objetos como registros de la base de datos, además de para otra serie de operaciones del front con las que todavía no estoy familiarizado.

Luego, sobre el atributo id, tengo otras dos notaciones, las cuales necesito para que JPA trabaje con los datos en la BD. La primera es @Id, para que

JPA identifique el id de una tabla. La segunda, @GeneratedValue, se usa para que JPA indique a la base de datos que este es un atributo autoincremental. Había oído hablar del patrón estrategia, pero no tenía ni idea de que uno de sus usos fuese este. Una cosa más que he aprendido gracias a esta actividad.

Después de esto defino el resto de atributos. Lo único reseñable es que fechaPublicacion lo defino como un long. Se generará de forma automática en el front y se almacenará tal cual, luego en el propio front ya me encargo de darle formato.

Nada más que explicar del modelo.

EL SERVICIO

```java
@Service
public class ServicioArticulos {

    @Autowired
    RepositorioDatos repositorioDatos;


    public List<Articulo> getArticulos() {
        return repositorioDatos.findAll();
    }

    public void guardarArticulo(Articulo articulo) {
        repositorioDatos.save(articulo);
    }
}
```

ILUSTRACIÓN 4 LA CLASE SERVICIO

En SpringBoot, para manejar los datos y llevar la lógica de la aplicación usamos los servicios. Ellos hacen de puente entre las clases Controller y las clases Repository, que son las que se encargan de comunicar los datos con la BD.

Para ello, lo primero que tenemos que hacer, es anotarlas como @Service para que SpringBoot identifique cual es su función. Creamos un objeto de la clase Repository, que veremos después, y lo anotamos también con @Autowired para que SpringBoot gestione su creación y destrucción cuando haga falta.

Luego defino dos métodos, estos mediante el objeto Repository se comunican con la BD. Gracias al modulo JPA, en vez de definir y usar los statements que usamos en SQL, usa sus propios métodos. En este caso tenemos findAll(), que es el equivalente a SELECT * FROM tabla, que nos devuelve una lista de todos los artículos, y luego tenemos save(articulo), que es el equivalente a

INSERT INTO VALUES. En este aspecto JPA nos facilita mucho las operaciones básicas. Otra cosa es cuando queremos hacer cosas más complejas. En ese caso, aunque no estoy seguro del todo, según lo que he visto por ahí, tendremos que hacerlas casi a la vieja usanza.

EL REPOSITORY

Según la estructura de SpringBoot, estas son las clases cuya única función es comunicarse con la base de datos, así que su definición como clase es muy sencilla.

```
@Repository
public interface RepositorioDatos extends JpaRepository<Articulo, Integer> {

}
```

ILUSTRACIÓN 5 LA INTERFAZ REPOSITORY

Como para el resto de clases, hay que añadir una notación al principio para que el framework sepa cual es la funcionalidad de esta clase. Y ahora viene lo que para mi era nuevo, pues no habíamos visto nada de esto en clase, por lo que tuve que buscar información por internet y youtube. Hay mucha gente que lo define como una clase normal y otros defendían que debe declararse como una interface, ya que en proyectos básico no se le va a definir ningún atributo y lo único que se va a dedicar es usar un patrón Singleton para comunicarse con la BD. Yo al final me decanté por la segunda opción, no por nada especial, simplemente porque me convenció más la explicación de los que defendían la creación de una interface.

Como última parte hay que hacer que esta interface extienda de JPARepository, para que pueda usar los métodos que realizan las operaciones habituales de una base de datos. Adicionalmente, debemos indicarle el tipo de

objeto o dato con el que va a trabajar, en este caso, Artículo. Con esto JPA ya va a crear una tabla artículo de no existir una y además va a crear columnas con el mismo nombre en la tabla. El segundo parámetro, Integer, le indica a JPA de que tipo es la clave primaria de la tabla.

Con esto queda terminada la explicación de la clase Repository.

ARCHIVOS DE CONFIGURACIÓN

En este caso sólo voy a mencionar dos, aquellos en los que he tenido que tocar cosas, para las cuales he tenido que buscar ejemplos y tutoriales.

Estos archivos son el application.properties y data.sql.

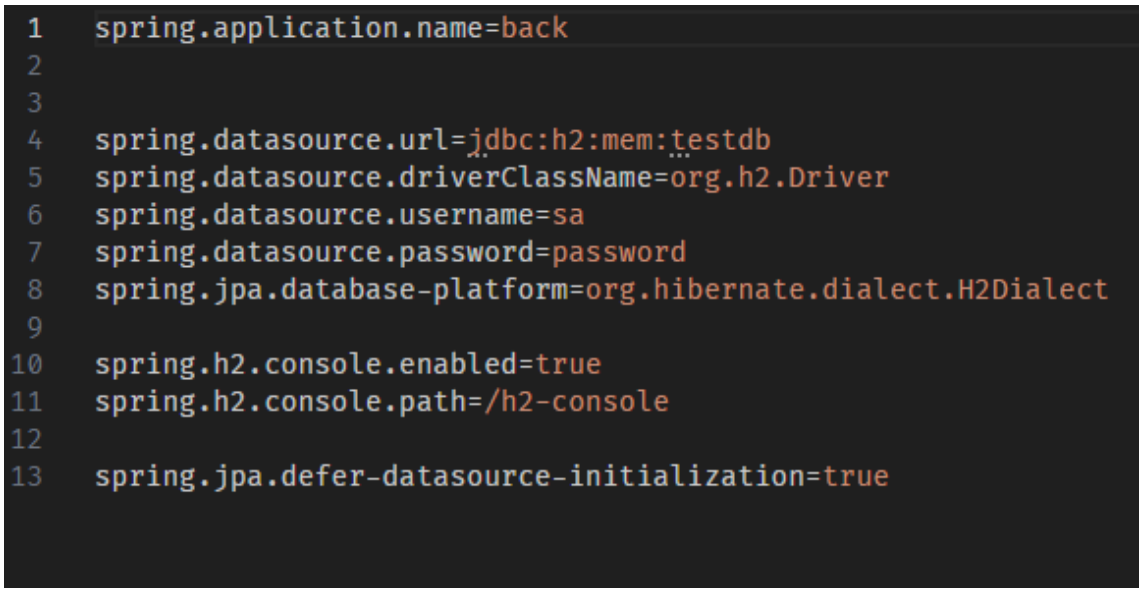


ILUSTRACIÓN 6 APPLICATION.PROPERTIES

En este archivo podemos incluir todos los parámetros de configuración que necesitásemos. En mi caso, todo lo que incluí fue para la configuración de la BD.

No voy a entrar mucho en detalles, pero resumiendo, los diferentes parámetros se refieren a la dirección y nombre de la base de datos, al tipo de

Driver que la va a manejar, el usuario y la contraseña, la plataforma y el lenguaje que usa, que se pueda acceder a la BD mediante la consola en el navegador, la dirección de la consola y, por último, que esta tarde un poco más en cargar para así cargar primero unos datos que le vamos a añadir mediante el fichero data.sql.

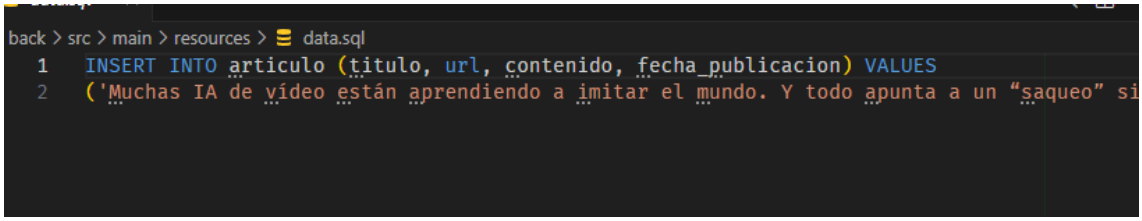


ILUSTRACIÓN 7 DATA.SQL

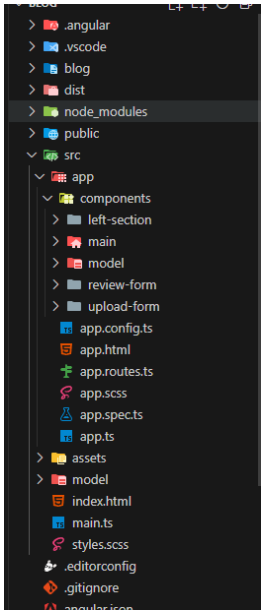
El segundo archivo es muy sencillo, es un insert básico de SQL para que, al arrancar el servidor, por lo menos exista un dato con el que se pueda trabajar.

Y hasta aquí la parte del back. Pasamos al front, que creo será más ligero de explicar.

FRONT EN ANGULAR V20

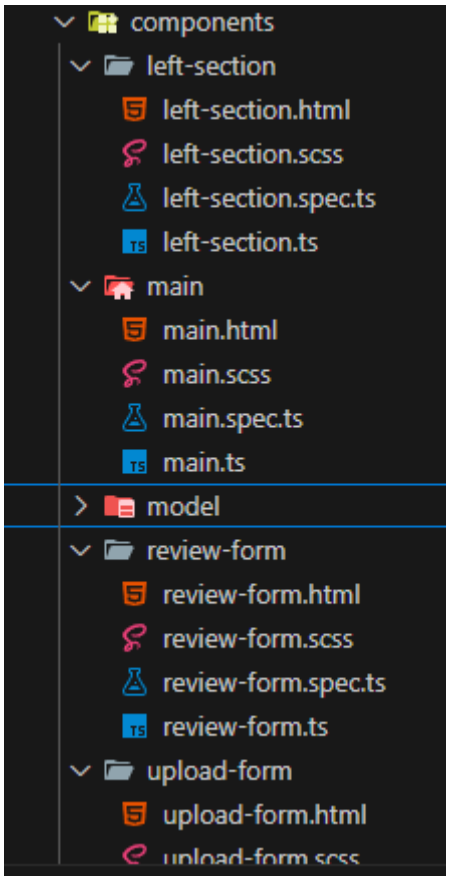
Como bien dije antes, el Front-End lo realicé en Angular, con sus pros, que me permite maquetar muy rápido una página en combinación con BootStrap, y sus contras, que recién había comenzado a aprender algo de él, y muchas de las cosas que he hecho han sido ensayo y error. Más de la segunda. Aquí la explicación espero sea más liviana. Me centraré en los detalles importantes.

EL ESQUELETO DE LA APLICACIÓN



Pues aquí el esqueleto de la parte del front. Es mucho más amplio en archivos que el back, ya que lleva muchos componentes y archivos de configuración pero la mayoría son componentes HTML con su clase propia para manejar los elementos que la componen. En mi caso, 4 componentes dentro de la carpeta components, una clase model equivalente a la clase model del back, pero en formato JSON para los objetos, una carpeta assets para las pruebas en las que tengo una colección y algunos métodos que una vez se conectaba el front al back quedó obsoleta, y otro carpeta model en la que hice pruebas con un segundo objeto en caso de que me diese tiempo a añadirle más funcionalidades a la aplicación pero que tuve que descartar por falta de tiempo, ya que todo lo anterior no conseguí que funcionase correctamente hasta casi el final del plazo.

LOS COMPONENTES



Estos son los componentes de la aplicación, es decir, cada bloque de la parte visual con su lógica. No vamos a entrar en detalle en todos ellos, pero para explicarlos un poco:

Left Section muestra el listado de artículos para poder elegir cual de ellos queremos leer.

Main es el contenedor principal. Incluye un NavBar en la parte superior y contiene a leftSection.

Model, ya lo dije antes, en la clase Artículo pero en Formato JSON.

Review-form iba a ser un formulario en el que la gente dejaba comentarios que se almacenarían en la base de datos pero que no me dio tiempo a desarrollar e implementar, pero simplemente no le eliminé, como a otros elementos que tuve que eliminar por ensayo y error.

Upload-form es el formulario que me permite insertar nuevos artículos en la BD.

Ahora simplemente voy a mostrar un par de capturas de ejemplo para que se pueda ver como codifiqué algunas de las partes.

```html
<div class="container-fluid m-2">
    <div class="row">

        <div class="col-12 col-md-4">
            @for(articulo of datosTemporales; track articulo.id){
            <div class="card w-90 mb-3" style="max-height: 300px">
                <div class="card-body overflow-hidden">
                    <h5 class="card-title">{{articulo.titulo}}</h5>
                    <p class="card-text">{{obtenerFecha(articulo.fechaPublicacion)}}</p>
                    <img [src]="articulo.url" [alt]="articulo.url">
                    <p class="card-text">{{articulo.contenido}}</p>
                </div>
                <a class="btn btn-primary w-50 m-auto mb-1" (click)="leerMas(articulo)">Leer Más</a>
            </div>
            }@empty{
                <p>No hay articulos</p>
            }
        </div>

        <div class="col-12 col-md-8">
            <app-left-section [articuloSeleccionado] = "articuloSeleccionado"></app-left-section>
        </div>
    </div>
</div>
```

ILUSTRACIÓN 8 CÓDIGO HTML DE MAIN

```typescript
async obtenerArticulo(){

    try {
        const response = await fetch('http://localhost:8080/api/articulos');
        if (!response.ok) {
            throw new Error('Error al cargar las tareas');
        }
        const data = await response.json();

        data.forEach((articulo : Articulo) => {
          this.datosTemporales.push(articulo);
          console.log(articulo);
        });

    } catch (error) {
        console.error('Error al cargar las tareas:', error);
    }


}



obtenerFecha(number : number){
    return new Date(this.articulo.fechaPublicacion).toLocaleDateString('es-ES');
}

leerMas(articulo : Articulo) {
  this.articuloSeleccionado = articulo;
  console.log('articulo enviado', this.articulo)
}
```

ILUSTRACIÓN 9 CÓDIGO TYPESCRIPT MAIN

Aquí una pequeña aclaración. Hay bibliotecas que facilitan las peticiones HTTP, pero como desconozco su forma de uso utilicé fetch, que es algo que si conocía.

```html
Go to component
<div class ="container d-flex flex-column w-75 mt-5">

    <form [formGroup]="formulario" (ngSubmit)="submitForm()">
        <div class="mb-3 d-flex flex-column">
            <label for="titulo" class="form-label">Titulo del Articulo</label>
            <input type="text" formControlName="titulo" class="form-control-lg" id="titulo" placeholder="Introduzc
        </div>
        <div class="mb-3 d-flex flex-column">
            <label for="url" class="form-label">Inserte URL de la imagen</label>
            <input type="text" formControlName="url" class="form-control-lg" id="url" placeholder="Introduzca el t
        </div>
        <div class="mb-3 d-flex flex-column">
            <label for="contenido" class="form-label">Contenido del artículo</label>
            <textarea class="form-control-lg" formControlName="contenido" id="contenido" rows="3" placeholder="Ins
        </div>
        <button type="submit" class="btn btn-primary" [disabled]="formulario.invalid">Subir</button>
    </form>
</div>
```

ILUSTRACIÓN 10 HTML DEL FORMULARIO

```typescript
  imports: [ReactiveFormsModule],
  templateUrl: './upload-form.html',
  styleUrl: './upload-form.scss',
})
export class UploadForm{

  formulario: FormGroup;

  constructor(){

    this.formulario = new FormGroup({

      titulo : new FormControl('', [Validators.required, Validators.minLength(10)]),
      url : new FormControl('', [Validators.required, Validators.minLength(10)]),
      contenido : new FormControl('', [Validators.required, Validators.minLength(10)]),
    })


  }


  async submitForm() {
    if(this.formulario.valid){

      const nuevoArticulo : Articulo = {
        id: 0,
        fechaPublicacion: Date.now(),
        titulo: this.formulario.value.titulo,
        url: this.formulario.value.url,
        contenido: this.formulario.value.contenido
      }

      //articulos.push(nuevoArticulo);
```

ILUSTRACIÓN 11 TYPESCRIPT DEL FORMULARIO

Otro pequeño apunte. En el TypeScript del formulario había validaciones para los campos, pero tuve que dejarlas sin uso por falta de tiempo para implementar el código HTML que refleja su funcionamiento.

COMENTARIOS SOBRE EL FRONT

Para realizarlo fui siguiendo el temario del curso, con ayuda de la documentación de www.angular.dev y perplexity o chatgpt sintetizando ambos contenidos. El problema es que estas IAs están entrenadas con versiones antiguas de Angular, por lo que mucha de su información o de sus ejemplos no son aplicables a este proyecto, que está realizado en la versión 20 de este framework. Por eso al final he consumido mucho tiempo en ensayo y error. Otro de mis fallos fue crear demasiados componentes para dividir la página en secciones individuales. Quería seguir la parte de buenas prácticas que hay en angular.dev y las indicaciones que suelen hacer en libros como clean code. El problema es que si no estás muy seguro de lo que haces eso es un problema.

Luego está el tema de la comunicación entre componentes. No conozco todavía nada de los objetos Observables ni de como usar correctamente los servicios, por lo tanto, tuve que usar métodos más básicos y, al haber creado muchos componentes, se me complicó mucho la tarea, por lo que tuve que ajustar constantemente y a veces me encontraba dando dos pasos hacia atrás para dar uno hacia delante. Por cosas como esta hay un componente que se llama left-section, que indica que hubo una parte izquierda de un todo en un inicio. Al final fui eliminando componentes y quedó ese con la información relevante y que funcionaba.

De todas formas, también probé cosas con rutas, para obtener un buen ejemplo de una SPA. En esa parte aprendí bastante, pero también tuve que reajustar y eliminar cosas. Por eso en el proyecto final la página principal es una pantalla en blanco y no un listado de artículos. Pero bueno, dejo más explicaciones para la presentación en Power Point que va a acompañar este documento.

CONEXIÓN DE LAS PARTES

Para conectar las partes, al final, ha sido de lo más sencillo, por decirlo de alguna manera. Como ya he explicado anteriormente en el documento, tenia un back con un BD, con sus endpoints definidos para marcar los puntos de conexión desde el front.

Luego, la parte del front, se comunicaba con el back mediante una función asyn de JavaScript. Es algo que ya hemos visto en clase y que se hacer.

Después de esto y de probarlo en local, quedaba la última parte. Esta última parte consistió en buscar información sobre como compilar y desplegar un proyecto empaquetado en .jar de SpringBoot y en hacer los mismo para el proyecto de Angular.

Ninguna de las dos supuso mucho problema, más allá de informarse sobre lo que había que hacer para desplegar las partes, además de conocer las opciones.

Para SpringBoot, tenia que compilar el proyecto usando Maven. Aquí viene la ventaja de los .jar y de SpringBoot. No hace falta desplegarlos en ningún tipo de servicio. Los .jar de SpringBoot tienen su propio servidor Tomcat embebido en su código, por lo que después de compilarlo sólo tuve que ejecutarlo por consola para levantar el servidor. Al tener también integrada la BD, al levantar este servicio, tenía levantados ambos. Así que, dos cosas menos.

Luego vino Angular. Parecido a lo que hemos hecho. Lo compilas, en el resultado va tu proyecto y una carpeta "dist" que es la que contiene los archivos que te interesan. Estos archivos son un HTML, que unifica a los que tienes en tus componentes, y varios archivos .js con las diferentes funcionalidades.

Como ya había oído acerca del servidor web Nginx, me informé para saber si podía desplegar este proyecto en el. Y sí, podía. Así que me dediqué a ello.

Descargué Nginx y su proceso de instalación es muy similar a lo que sería Apache. Descomprimes el .rar donde quieras, en mi caso C:. Dentro de esa carpeta hay una carpeta llamada "HTML" equivalente al "HTDOCS" del servidor Apache. Lo único que tuve que hacer fue copiar el contenido de la carpeta "dist" dentro y tocar un par de líneas de un archivo de configuración relativas al puerto a usar y la dirección por defecto.

Después sólo debo de ejecutar mi archivo .jar mediante Power Shell, y el servidor Nginx mediante "cd .\nginx.exe" para levantar el servicio web. Y todo se ejecutó de forma correcta.

No incluyo capturas de esta parte por varios motivos. El primero es por el Power Point que voy a adjuntar con este documento. El siguiente es que las capturas iban a ser o bien de descomprimir y copiar archivos o bien de ejecutar comandos por terminal, que también lo incluyo en la presentación. El último es porque creo que es algo muy similar a cosas que ya hemos hecho y, basándome en lo que he hecho yo, me pareció bastante sencilla la parte final, no creo que merezca la pena añadir más capturas de pantalla.

PARTE MÁS DIFÍCIL Y SOLUCIÓN

Como ya he dicho antes, la parte más difícil fue desarrollar el front. A parte de que mi sentido del gusto está algo atrofiado y mis diseños, por decirlo de alguna forma, son un tanto ortopédicos, el hecho de usar algo en lo que estoy empezando añadió mucha dificultad.

Con esto dicho, que no se entienda que me habría sido más fácil ni que el resultado habría sido mejor si lo hubiese hecho en HTML, CSS y JS, pero lo de aprender sobre la marcha esta vez se me ha atragantado un poco.

La forma en la que los componentes de Angular se comunican entre sí, si por comunicarse entendemos que las formas que conozco de hacerse son muy básicas, y algunos otros aspectos como los imports de unos componentes a otros ha hecho que tuviese que estar constantemente rectificando, eliminando o volviendo a crear algunas partes.

Luego, por último, el hecho de entender que el despliegue se tuviese que hacer en una máquina virtual, que enlentece mucho las cosas. Además, la máquina virtual que tenía para este módulo no funcionaba correctamente, no se si es por que no estaba activada o algo así. Tuve que volver a instalar Temurín 17, Maven, añadir las Paths de sistema. Al final descargar e instalar Nginx fue de los más sencillo que tuve que hacer.

Al final, si me pongo a mirarlo bien, esto lo resolví por pura cabezonería, ni más, ni menos. Un fallo de cálculo el pensar que en la semana de vacaciones iba a ser capaz de aprender lo necesario de Angular como para que fuese un resultado solido y un desarrollo placentero. Pero bueno, así también se aprende.

OTROS SERVIDORES WEB Y DIFERENCIAS CON APACHE

Pues en mi caso es bien fácil. Nginx que es el que he usado para esta práctica.

Por el uso que le he dado en este ejercicio, voy a indicar una diferencia importante. Si no recuerdo mal, Apache cuenta con un apartado gráfico, por navegador. Sin embargo, Nginx, o no lo tiene, o no he sabido encontrarlo. Con apache cuando cargas localhost puedes encontrar algo, con Nginx solo te da la bienvenida en el navegador. Y ya.

Buscando por internet he encontrado otras comparativas bastante más importantes que la mía:

Nginx es más eficiente que Apache a la hora de trabajar con contenido estático.

Apache trabaja de forma nativa el contenido dinámico, Nginx necesita ayuda o servicios externos.

Apache puede cargar módulos o extensiones de forma dinámica, Nginx tiene que ser recompilado antes de hacer efectivo el uso de estos módulos.

Nginx es mucho más escalable que Apache.

Apache consume muchos recursos bajo altas cargas de trabajo. En ese aspecto Nginx es más eficiente.

Apache es más flexible, gracias a que no tiene que recompilarse, la gran cantidad de módulos y la forma en la que trabaja con contenido dinámico. En este punto, Nginx es más rígido y limitado.

En conclusión, Apache brilla en trabajos dinámicos y flexibles, mientras que Nginx lo hace en entornos con gran cantidad de tráfico y en proyectos muy grandes.