MTIS-Practica2

Autor: Ramsés Martínez Martínez

Práctica 2 de la asignatura de Metodologías y Tecnologías de Integración de Sistemas

Introducción

Una de las tareas más importantes a la hora de crear una SOA es definir su modelo de servicios: o sea, qué servicios hay y qué tareas en concreto hace cada uno. En una SOA basada en servicios web tipo Rest debemos de poder diseñar nuestros servicios al igual que hacíamos con los servicios web tipo SOAP mediante los contratos WSDL, para poder definir en un documento todas las funcionalidades que presenta nuestra API Rest.

RESTful API Modeling Language (RAML) hace que sea fácil administrar todo el ciclo de vida de la API, desde el diseño hasta el uso compartido. Es conciso, solo se escribe lo que se necesita definir, y es reutilizable. Es un diseño API legible por máquina y amigable para los humanos.

El Objetivo de esta práctica es documentar un servicio mediante el RAML, sin necesidad de crear un servicio web. Posteriormente crearemos un servicio web basándonos en el documento RAML y por último generaremos un cliente.

Componentes

En esta apartado vamos a ver los **distintos componentes de la práctica**, tanto el servidor, como el cliente, así como la documentación generada con RAML.

Cabe destacar, que a diferencia de la documentación de la práctica anterior, en esta ocasión, no nos vamos a centrar en mostrar el método, ni tampoco tendremos capturas de segmentos de código, salvo en alguna aclaración puntual, que tal vez si necesitemos un segmento del código original.

Servidor API REST (NodeJS)

Hemos decidido realizar nuestra **API REST con NodeJS**, puesto que es un entorno en tiempo de ejecución multiplataforma, de codigo abierto. Además, lo hemos utilizado en asignaturas recientes, por lo que, nos viene bien para afianzar conocimientos.

Para la realización de la API, **nos hemos basado en una estructura básica** de proyectos, implementando la capa de preparado y conexión de la API, en el fichero "app.js". Además, hemos utilizado un único archivo para almacenar todas las rutas de nuestra API, ya que, en nuestro caso, **no tenemos más de 4 peticiones**, por lo tanto, podemos tenerlas todas en un mismo archivo sin llegar a ser muy extenso.

En cuanto a los distintos servicios ofrecidos, hemos implementado un modelo para cada uno de los servicios (NIF, IBAN, Código Postal y Generar Presupuesto), donde en dichos modelos, nos encargamos de conectar con nuestra base de datos, que en nuestro caso, es **exáctamente igual que la de la práctica anterior** y que tenemos el esquema en este mismo directorio "practica2.sql" **realizada en MySQL**, en concreto, usando **XAMPP**

En algunos casos, hemos tenido que utilizar estados de respuesta **200** pese a que realmente podríamos contestar con un **400** para indicar que la petición ha sido realizada de manera incorrecta o incluso con un **401**, para indicar que no se tiene acceso, puesto que la API KEY puede ser errónea, pero, puesto que en el enunciado nos indica claramente, que en algunos casos, tenemos que responder con algún tipo de mensaje según el error, hemo tenido que utilizar el **OK** pese a que realmente no es una petición correcta.

```
app.get("/api/validarIBAN/:iban", function (req, res) {
  var iban = req.params.iban;
  var restKey = req.get('restKey');
  if(iban) {
    IBANModel.validarIBAN(iban, restKey, function (error, data) {
      if (error) {
        res.status(200).json({ existe: data, message: error });
      } else {
        res.status(200).json(data);
      }
    });
  }
  else {
    res.status(400).json({"msg":"El IBAN no tiene el formato correcto"});
  }
});
```

Como podemos observar, en **caso de detectar un error** deberíamos en este caso, dependiendo del error, mandar un **código de estado** u otro, pero hemos tenido que mandar un 200, para que en el cliente, podamos procesar de manera más sencilla el contenido.

Cliente API REST (Angular2)

En nuestro caso, hemos decidido **utilizar un cliente web** y para ello, nos hemos decantado por usar **Angular 2**, puesto que nos apeteciía experimentar un poco más con nuestros conocimientos prácticamente escasos de dicho **framework** y aprender una nueva manera de ver el mundo web.

En nuestro caso, hemos empezado generando el proyecto, mediante los comandos que nos proporciona **@angular/cli** habiendo instalado dicho paquete mediante **npm**.

Además, hemos utilizado para el estilo de todo nuestro proyecto **Bootstrap 4**, para poder estilizar nuestro html, para que sea más visible y conseguir una interfaz más intuitiva.

Hemos utilizado distintos tipos de herramientas y servicios que nos ofrece **Angular 2** como los componenetes, servicios, interfaces y modelos, pero lo más significativo, quizá, es la forma en que llamamos a nuestra API. Podemos ver un ejemplo, que en este caso, corresponde a la llamada para **verificar el NIF**, ubicado en el archivo "servicios.service.ts"

```
apiURL: string = "http://localhost:3000/api"
...
validarNIF(nif: string) {
```

```
    let requestURL = `${this.apiURL}/validarNIF/${nif}`
    let headers = new Headers({
      'restKey':'soapkeydeprueba12345678'
    });

    return this.http.get(requestURL, { headers })
      .map(res => res.json());
  }
```

Como podemos observar, hemos utilizado **cabeceras** para la introducción manual de la clave de nuestra API, puesto que, según nuestro entendimiento, las **claves** suelen generarse y asignarse mediante **tokens** a la hora de **iniciar sesión** o **registrarse**.

Documentación RAML

La documentación, la hemos generado, utilizando **API Designer** que es la herramienta que se nos propone en la práctica, para no únicamente genrar documentación, si no, incluso podríamos probar dicha documentación e incluso generar un cliente.

La documentación está dentro de la carpeta del mismo nombre.

```
..\MTIS-Practica2\Documentacion
```

Despliegue

En este apartado, vamos a explicar la forma de desplegar el proyecto, para el correcto funcionamiento del mismo.

En primer lugar, deberíamos arrancar el **XAMPP**, en concreto, los servicios de **APACHE** y **MySQL**, una vez arrancado, **podremos dirigirnos a nuestro navegador favorito** para acceder a **phpmyadmin** donde **importaremos** el esquema **SQL**.

Una vez importado el esquema SQL, procederemos a arrancar el servidor. Para ello, tendremos que instalar toda la paquetería, mediante:

```
npm install
```

Todo esto, estando **dentro** del directorio del **servidor**. Tras instalar los paquetes y las dependencias, podremos iniciar el servidor, mediante:

```
node app.js
```

o

```
nodemon app.js
```

Por último, podremos arrancar nuestro cliente, **entrando en el directorio del cliente** e instalando los paquetes y dependencias mediante:

```
npm install
```

Una vez instalado, podremos proceder a arrancar el cliente, mediante:

```
ng serve -o
```

De esta manera, se nos abrirá automáticamente una ventana en el navegador con nuestro cliente listo para ser probado.