IMPLEMENTAR UN CLIENTE REST CON SPRING: RESTTEMPLATE

Spring nos provee de un cliente Rest, RestTemplate, muy sencillo de usar. El se encarga de realizar la conexión http, lo único que hace falta es pasarle la url del servicio contra el que conectar. El mismo RestTemplate gestiona sus propios 'messageConverters', con los que parsear los datos enviados y recibidos de/a json, por ejemplo. Solo es necesario añadir en nuestro proyecto la dependencia necesaria (Jackson, en nuestro caso).

En el ejemplo veremos como llamar al servicio con los métodos básicos (GET, POST, PUT, DELETE). Para hacerlo más sencillo, lo haremos directamente desde un programa java desde su método 'main'.

Nos creamos un proyecto maven básico (por ejemplo, con el archetype 'maven-archetype-quickstart'), y añadimos las siguientes dependencias:

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-web</artifactId>
        <version>4.1.6.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>2.5.3</version>
    </dependency>
    ...
</dependencies>
```

La dependencia de 'spring-web' es precisamente para poder tener acceso a la clase RestTemplate. La 'jackson-databind' es necesaria para el parseo de datos a json.

Creamos una clase con método main, en la que implementaremos los ejemplos:

AppBookRestApiary.java

```
package com.edwise.pocs.springrestclient;

public class AppBookRestApiary {

    private static final String URL_API_BOOKS =
            "http://private-114e-booksapi.apiary-mock.com/books/";
```

```
7
8       public static void main(String[] args) {
9
10      }
11
12  }
```

Tendremos la url a nuestro servicio (http://private-114e-booksapi.apiary-mock.com/books/) directamente en una constante.

Creamos una clase para la entity que nos devuelve el servicio:

Book.java

```
1  package com.edwise.pocs.springrestclient.model;
2
3  public class Book {
4      private Long id;
5      private String title;
6
7      public Long getId() {
8          return id;
9      }
10
11     public void setId(Long id) {
12         this.id = id;
13     }
14
15     public String getTitle() {
16         return title;
17     }
18
19     public void setTitle(String title) {
20         this.title = title;
21     }
22
23     @Override
24     public String toString() {
25         return "Book{" +
26                 "id=" + id +
27                 ", title='" + title + '\'' +
28                 '}';
29     }
30 }
```

Necesitamos este bean o entity para recuperar los datos, etc. Es tan sencillo como revisar el servicio, ver sus campos e implementar una clase java con esos campos. Jackson se encargará del parseo a/desde json.

Creamos nuestro objeto RestTemplate, con el que accederemos al servicio. Al crearse, por defecto ya tiene los converters necesarios para el parseo a/desde json, al tener como dependencia la librería Jackson.

```
public static void main(String[] args) {
    RestTemplate restTemplate = new RestTemplate();

}
```

Es así de simple. Tiene también disponible algún otro constructor, con parámetros, como los converters, etc. Pero con el constructor sin parámetros nos sobra.

Ahora probaremos cada uno de los métodos que acepta el servicio:

– GET all: para obtener todos los 'books':

```
ResponseEntity<Book[]> response =
            restTemplate.getForEntity(URL_API_BOOKS, Book[].class);

System.out.println();
System.out.println("GET All StatusCode = " + response.getStatusCode());
System.out.println("GET All Headers = " + response.getHeaders());
System.out.println("GET Body (object list): ");
Arrays.asList(response.getBody())
            .forEach(book -> System.out.println("--> " + book));
```

Tanto para get como para post, tenemos disponible dos tipos de métodos: **getForObject/postForObject** y **getForEntity/postForEntity**. Los segundos te devuelven una respuesta completa (es el que uso en el ejemplo), en la que poder obtener tanto headers, como código http devuelto, etc. Los 'xxxForObject' devuelven directamente el objeto (el body de la respuesta).

En este caso, le tenemos que pasar la url y el tipo esperado para el body (array de clase 'Book' en nuestro caso). Mostramos luego el código http, los headers y, en bucle, todos los 'books' devueltos. El método **'getBody'** nos devolverá un 'Book[]', al haberlo parametrizado así en la llamada al get.

– GET: para obtener un 'book' en concreto, por su id:

```
ResponseEntity<Book> response =
            restTemplate.getForEntity(URL_API_BOOKS + "{id}",
Book.class, 12L);

```

```
System.out.println();
System.out.println("GET StatusCode = " + response.getStatusCode());
System.out.println("GET Headers = " + response.getHeaders());
System.out.println("GET Body (object): " + response.getBody());
```

Similar al caso anterior. En este caso, le pasamos **Book.class** como tipo esperado.

Por otro lado, el tercer parámetro serían las 'pathvariables' del endpoint. En este caso el id.

– POST: para insertar un 'book' nuevo:

```
Book bookToInsert = createBook(null, "New book title");
ResponseEntity<Book> response =
            restTemplate.postForEntity(URL_API_BOOKS, bookToInsert,
Book.class);

System.out.println();
System.out.println("POST executed");
System.out.println("POST StatusCode = " + response.getStatusCode());
System.out.println("POST Header location = " +
response.getHeaders().getLocation());
```

Creamos un nuevo 'Book', sin id, y lo enviamos por post, con un 'postForEntity'. En el segundo parámetro le pasamos lo que seria el 'body' de la request (nuestro objeto) y en el tercer parámetro el tipo.

En este caso no mostramos el body de la respuesta, dado que lo único que nos devuelve es el 'location'

– PUT: para actualizar un 'book' en concreto, por su id:

```
Book bookToUpdate = createBook(123L, "Book title updated");
restTemplate.put(URL_API_BOOKS + "{id}", bookToUpdate, 123L);

System.out.println();
System.out.println("PUT executed");
```

Aquí cambia un poco la cosa. Excepto para get y post, para el resto de métodos http, RestTemplate nos ofrece unos métodos más simples, demasiado en mi opinión. Como vemos en este caso, el método **'put'** no devuelve nada. Si necesitáis poder comprobar el código http, los headers o algún otro dato de la respuesta, echadle un vistazo al

método ‘exchange’. Es un método con el que podemos realizar cualquier tipo de llamada (GET, POST, etc).

– DELETE: para borrar un ‘book’ en concreto, por su id:

```
restTemplate.delete(URL_API_BOOKS + "{id}", 12L);

System.out.println();
System.out.println("DELETE executed");
```

Igual que el caso de put