


Open Opened 2 days ago by  **Rubén Montero**

GET /movies

Resumen

- Crearemos una nueva clase `MoviesREST`
- Uniremos los dos grandes conceptos trabajados hasta ahora:
 - **APIs REST**
 - **Conectores a bases de datos**
- Serviremos en un nuevo *endpoint* `/movies` los datos de las películas en `movies.db`

Descripción

Hemos trabajado con dos conceptos esenciales durante este proyecto: APIs REST y conectores JDBC para bases de datos.

¿Es posible unirlos? ¿Tiene sentido? ¿Cómo?

[Uniendo dos conceptos](#)

Las respuestas son **sí, sí, y así**:

Comencemos creando una nueva clase `MoviesREST.java`. Será similar a `SimpleREST.java` pero estará destinada a ofrecer un API REST que por detrás trabaja contra la base de datos SQLite `movies.db`:

```
import org.restlet.Component;
import org.restlet.data.Protocol;
import org.restlet.routing.VirtualHost;

public class MoviesREST {
    private Component component;

    public void runServer() {
        try {
            this.component = new Component();
            this.component.getServers().add(Protocol.HTTP, 8104);
            VirtualHost host = this.component.getDefaultHost();
            // Aquí mapearemos los endpoints
            // ...
            this.component.start();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    public void stopServer() throws Exception {
        if (this.component != null) {
            this.component.stop();
        }
    }
}
```

Como puedes ver, también escucha en el puerto TCP 8104, por lo que no es posible que una instancia de `MoviesREST` y una de `SimpleREST` estén activas simultáneamente.

Un primer *endpoint*: `/movies`

Empecemos por el POJO

Añadamos un nuevo POJO `Movie.java`, con los **atributos** que queramos enseñar desde nuestra API REST. Usaremos los mismos que hay disponibles en la base de datos:

```
public class Movie {
```

```

private int id;
private String title;
private int year;
private int duration;
private String countryIso3166;
private String genre;
private String synopsis;
}

```

Ahora, podemos darle a **Code > Generate > Constructor** para crear automáticamente el método constructor *con todos* los atributos.

Para terminar el POJO, crearemos un método `toJSONObject` que *serialice* a `JSONObject` **todos** los atributos, usando como *clave* siempre el mismo *nombre* del atributo:

```

public JSONObject toJSONObject() {
    JSONObject object = new JSONObject();
    object.put("id", this.id);
    object.put("title", this.title);
    // Resto de atributos
    // ...

    return object;
}

```

Clase `ServerResource`

Ahora, **añadamos** una nueva subclase de `ServerResource`.

La llamaremos `GetAllMovies.java`:

```

import org.restlet.representation.StringRepresentation;
import org.restlet.resource.Get;
import org.restlet.resource.ServerResource;

public class GetAllMovies extends ServerResource {

    @Get
    public StringRepresentation getEndpointResponse() {
        // ...
    }
}

```

Ahora la idea es que en `getEndpointResponse`, nos **conectemos** a la base de datos, lancemos un `SELECT` y usemos el resultado.

¿Podemos escribir todo el código aquí, en `getEndpointResponse`?

Sí.

¿Lo haremos?

No.

Vamos a usar una solución más elegante, aprovechando el `MoviesConnector.java` de tareas anteriores.

Nuevo método en `MoviesConnector`

Añadamos a dicha clase un nuevo método que devuelva una *lista de Movies*:

```

public ArrayList<Movie> getAll() {

}

```

¿Recuerdas cómo usábamos `this.connection` en dicha clase para crear un `statement` y lanzar una consulta? ¿Recuerdas la diferencia entre `executeUpdate` y `executeQuery`?

Nosotros lanzaremos un `SELECT * FROM TMovies` así:

```

public ArrayList<Movie> getAll() {
    try {
        Statement statement = this.connection.createStatement();
        String sql = "SELECT * FROM TMovies";
        ResultSet result = statement.executeQuery(sql);
        // En 'result' están los resultados de la consulta SQL
        // ...
        statement.close();
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}

```

Y ahora, pondremos en práctica un bucle `while` donde *instancia* una nueva `Movie` por cada película, y se se *acumula* en un `ArrayList` :

```

public ArrayList<Movie> getAll() {
+   ArrayList<Movie> allMovies = new ArrayList<>();
    try {
        Statement statement = this.connection.createStatement();
        String sql = "SELECT * FROM TMovies";
        ResultSet result = statement.executeQuery(sql);
+       while(result.next()) {
+           int id = result.getInt(1);
+           String title = result.getString(2);
+           int year = result.getInt(3);
+           int duration = result.getInt(4);
+           String countryIso3166 = result.getString(5);
+           String genre = result.getString(6);
+           String synopsis = result.getString(7);
+           // Instanciamos una nueva Movie con los valores de la fila
+           Movie aMovie = new Movie(id, title, year, duration, countryIso3166, genre, synopsis);
+           allMovies.add(aMovie);
+       }
        statement.close();
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
+   return allMovies;
}

```

¡Listo!

Completando `ServerResource`

Ya estamos en condiciones de terminar la clase `GetAllMovies.java` que comenzamos anteriormente.

Sólo debemos...

1) Instanciar un `MoviesConnector`

```

@Get
public StringRepresentation getEndpointResponse() {
    MoviesConnector connector = new MoviesConnector();
    // ...
}

```

2) Instanciar un `JSONArray`

Se trata del *array* JSON (`[]`) que enviaremos como respuesta HTTP.

De momento, está vacío.

```

@Get
public StringRepresentation getEndpointResponse() {
    MoviesConnector connector = new MoviesConnector();
    JSONArray resultArray = new JSONArray();
    // ...
}

```

3) Invocar el `connector.getAll()` que creamos anteriormente

Y almacenamos el resultado en una variable tipo `ArrayList<Movie>`

```
@Get
public StringRepresentation getEndpointResponse() {
    MoviesConnector connector = new MoviesConnector();
    JSONArray resultArray = new JSONArray();
    ArrayList<Movie> databaseFilms = connector.getAll();
    // ...
}
```

4) Para cada resultado, lo añadimos al *array* JSON

Usamos un bucle `for-each` por su sencillez, aunque un bucle `for` convencional también sería posible.

```
@Get
public StringRepresentation getEndpointResponse() {
    MoviesConnector connector = new MoviesConnector();
    JSONArray resultArray = new JSONArray();
    ArrayList<Movie> databaseFilms = connector.getAll();
    for (Movie mov : databaseFilms) {
        resultArray.put(mov.toJSONObject());
    }
    // ...
}
```

5) Finalmente, devolvemos todo como un `StringRepresentation`

```
@Get
public StringRepresentation getEndpointResponse() {
    MoviesConnector connector = new MoviesConnector();
    JSONArray resultArray = new JSONArray();
    ArrayList<Movie> databaseFilms = connector.getAll();
    for (Movie mov : databaseFilms) {
        resultArray.put(mov.toJSONObject());
    }
    connector.closeConnection(); // No olvidemos cerrar la conexión
    String jsonString = resultArray.toString();
    StringRepresentation representation = new StringRepresentation(jsonString);
    representation.setMediaType(MediaType.APPLICATION_JSON);
    return representation;
}
```

¡Listo!

Basta con *mapear* el *endpoint* adecuadamente en `MoviesREST.java` :

```
- // Aquí mapearemos los endpoints
- // ...
+ host.attach("/movies", GetAllMovies.class);
```

Probando y saliendo victoriosos

Si instancias y ejecutas tu nuevo servidor en `Main.java` :

```
- SimpleREST myServer = new SimpleREST();
- myServer.runServer();
+
+ MoviesREST server = new MoviesREST();
+ server.runServer();
```

y le das a **Play**, ya podrás visitar `http://localhost:8104/movies` desde tu navegador.

Estás observando los **datos** de la **base de datos SQLite**.

¿Qué significa esto?

Que has desarrollado tu primera API REST que **realmente** consume datos de una base de datos.

¡Enhorabuena! Es un paso muy importante.

Ahora, la base de datos podría modificarse y tu servidor, sin detenerse y relanzarse, mostraría los datos **actualizados** a las **nuevas peticiones**.

Así es como funcionan las cosas en el mundo de las aplicaciones distribuidas.

Por último

Verifica que el `test` funciona correctamente.

Haz `commit` y `push` para subir los cambios al repositorio.



[Rubén Montero @ruben.montero](#) changed milestone to [%Sprint 2](#) 2 days ago