


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

REST APIs

Resumen

- Introduciremos el concepto de API REST
- Veremos cómo están formateados las peticiones y respuestas HTTP
- Hablaremos del concepto *endpoint*
- Usaremos la librería *Restlet* para servir un *endpoint* REST sencillo, escribiendo las clases `SimpleGetExample` y `SimpleREST`

Descripción

Acceder directamente a los datos de una base de datos conlleva establecer conexiones como las que hemos empleado en tareas anteriores. Pero, ¿**todo el mundo** se conecta a las bases de datos?

¿Todo el mundo?

En el mundo de las aplicaciones distribuidas, es importante conocer **qué** es la **arquitectura cliente-servidor**. Seguramente es algo con lo que ya somos familiares. La idea es que las aplicaciones **cliente** (e.g.: navegador web, *app* móvil,...) se conectan a una máquina **servidor** que atiende **peticiones**.

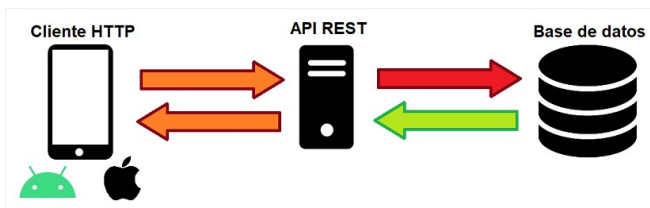
La respuesta es no

No *todo el mundo* accede 'de la misma manera' a los datos de una base de datos.

Por lo general, **no** se quiere exponer una base de datos **directamente al público**. La arquitectura predominante en la actualidad es la de [API REST](#).

¿Qué es un API REST?

En esencia, se trata de una *fachada* empleada para exponer los datos de una base de datos.



Conceptos esenciales

HTTP

En una API REST se sirven los datos que piden los clientes.

Al igual que cuando se navega a una página web, el protocolo mediante el cual se atienden las peticiones es [HTTP](#).

Podemos entender que HTTP es una **autovía**. Una **red de carreteras**, digamos.

Se concibió para que **el mundo de la web** funcionase. Es decir, por esas **carreteras** circulan camiones **cargados de páginas web** y solicitudes de navegadores.

Nosotros, desarrollando un **API REST**, *aprovechamos* esa infraestructura de carreteras para poner a circular unos *ligeros y rápidos taxis* que se dedicarán a llevar *datos*. Más concretamente, trabajarán con JSON (en lugar de con pesados ficheros HTML y CSS).

Peticiones HTTP

Sea un camión o un taxi lo que circule, todo se pone en marcha cuando un *cliente* efectúa una *petición HTTP*.

Las peticiones tienen **3** partes principales:

- **Línea de petición.** En la primera línea se especifica el [verbo](#) (ó método) HTTP y el *recurso* solicitado. Algo así:

```
GET /publicaciones
```

Los verbos más usados son GET, POST, PUT, DELETE

- **Cabeceras.** Son pares clave-valor que especifican información adicional.

```
User-agent: Mozilla Firefox
Accept: application/json
```

- **Cuerpo.** Viene después de una línea en blanco e indica información adicional que se manda al servidor. Por ejemplo, si estamos usando una petición POST para crear una nueva 'publicación', puede que la información del usuario vaya en formato JSON:

```
{
  'texto_publicacion': '¡Estoy trasteando con APIs REST!'
}
```

Respuestas

También tienen **3** partes principales:

- **Línea de respuesta:** Contiene un [código de respuesta](#). ¿Te suena el típico *404 Not Found*? Es un código de respuesta HTTP. Se dividen en 5 categorías: 1xx Información, 2xx Éxito, 3xx Redirección, 4xx Error cliente, 5xx Error servidor.
- **Cabeceras:** Pares clave-valor enviados en la respuesta. Cosas como las famosas *cookies* viajan en las cabeceras de petición/respuesta
- **Cuerpo:** Donde el API REST responde información relevante. Nos centraremos en el formato JSON

El objetivo

Nosotros queremos **implementar un API REST**. Como la máquina central en la imagen de arriba.

Para ello, debemos **implementar endpoints**. Y, para ello, debemos saber *qué* son.

¿Qué es un *endpoint*?

Viene a ser una *ruta* que nuestro **servidor** está preparado para atender.

En el ejemplo anterior:

```
GET /publicaciones
```

`/publicaciones` es un *endpoint*.

Los *endpoints* pueden contener partes *variables* (*path params*) y parámetros después de un signo de interrogación `?` (*query params*). Los veremos más adelante.

¿Cómo implemento un *endpoint*?

Restlet

Usaremos la librería Restlet, un *framework* de código abierto que emplea clases y anotaciones. Es similar al popular [Spring Boot](#), pero mucho más sencillo (`org.restlet.jar` de 699kb vs. ~70Mb de librerías Spring).

Dispone de [documentación](#) con varios ejemplos, aunque lo que necesitemos para nuestros sencillos casos de uso lo veremos enteramente aquí.

Un sencillo **GET**

Vamos a implementar un *endpoint*. Lo lanzaremos en nuestra máquina local y atenderá la petición dirigida al *endpoint* `/example` en el puerto `8104`.

Abre tu navegador web preferido (e.g.: Mozilla, Chrome).

Escribe en la barra de direcciones:

```
http://localhost:8104/example
```

Dale a ENTER.

No se ve nada, ¿verdad? Es normal que falle al cargar. **No** hay un servidor que responda.

Fabricando un servidor REST

Primer paso: Una clase `ServerResource`

Cada *endpoint* se gestiona con una clase Java.

Nosotros añadiremos una clase llamada `SimpleGetExample`. El nombre *da igual*. Lo que importa es que **extienda** (*herede*) de `ServerResource`, que es una clase de la librería Restlet.

Esto de la *herencia* lo hemos mencionado con anterioridad. Basta con aclarar que debemos usar la palabra `extends`, así:

```
import org.restlet.resource.ServerResource;

public class SimpleGetExample extends ServerResource {

}
```

Ya tienes **tu subclase** de `ServerResource`.

Ahora, necesitamos **un método**. Debe estar anotado con `@Get` y llamarse `toString()`. Devolverá un `String`.

```
import org.restlet.resource.Get;
import org.restlet.resource.ServerResource;

public class SimpleGetExample extends ServerResource {

    @Get
    public String toString() {

    }

}
```

¿Y este método para qué?

Sirve para que indiquemos *qué* respuesta dará el servidor a la petición.

En concreto, Restlet cogerá el `String` que *devolvamos* y se encargará de entregarlo al cliente HTTP, por ejemplo, tu navegador web.

Devolvamos el siguiente mensaje:

```
public class SimpleGetExample extends ServerResource {

    @Get
    public String toString() {
        return "Hola, caracola";
    }

}
```

Segundo paso: Un `Component`

A continuación, añadiremos una nueva clase `SimpleREST.java`. No es estrictamente obligatorio (podríamos usar `Main.java` directamente), pero **lo haremos** para encapsular el servidor y escribir buen código.

```
public class SimpleREST {

}
```

Escribamos un método `runServer`. El objetivo es que **cuando se invoque**, el **servidor REST se ponga en marcha**.

Como primer paso, *instanciaremos* un objeto `Component` de la librería Restlet. Así:

```
import org.restlet.Component;

public class SimpleREST {
    private Component component;
```

```

    public void runServer() {
        this.component = new Component();
        // ...
    }
}

```

En este objeto `Component` podemos indicar el **protocolo** y **puerto** donde se atenderán las peticiones.

Por defecto, suele navegar en Internet en el puerto 443 con protocolo HTTPS. En caso de una conexión *no* cifrada, será HTTP y puerto 80.

Nosotros *serviremos* nuestro API REST mediante HTTP sin cifrar, y en el puerto arbitrariamente escogido 8104:

```

import org.restlet.Component;
import org.restlet.data.Protocol;

public class SimpleREST {
    private Component component;

    public void runServer() {
        this.component = new Component();
        this.component.getServers().add(Protocol.HTTP, 8104);
        // ...
    }
}

```

Mapear endpoints

A continuación, especificaremos *Quiero que la clase SimpleGetExample se encargue de atender peticiones al 'endpoint' /example*

Para ello, recurrimos a una clase propia de la librería, llamada `VirtualHost` :

```

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

public class SimpleREST {
    private Component component;

    public void runServer() {
        this.component = new Component();
        this.component.getServers().add(Protocol.HTTP, 8104);
        VirtualHost host = this.component.getDefaultHost();
        // ...
    }
}

```

...y sobre ese *host*, invocamos `attach` . Se encarga de hacer justo lo que queremos, **mapear** un *endpoint* a una **clase**:

```

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

public class SimpleREST {
    private Component component;

    public void runServer() {
        this.component = new Component();
        this.component.getServers().add(Protocol.HTTP, 8104);
        VirtualHost host = this.component.getDefaultHost();
        host.attach("/example", SimpleGetExample.class);
        // ...
    }
}

```

(Fíjate que se usa `.class` . Esto sirve en Java para representar una clase pero sin instanciarla. Será Restlet quien use dicha información para instanciar nuestra clase, a su manera)

Por último será necesario invocar `.start()` . Controlaremos la posible excepción relanzándola:

```
public class SimpleREST {
    private Component component;

    public void runServer() {
        try {
            this.component = new Component();
            this.component.getServer().add(Protocol.HTTP, 8104);
            VirtualHost host = this.component.getDefaultHost();
            host.attach("/example", SimpleGetExample.class);
            this.component.start();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

Tercer paso: Que se pueda detener

Vamos a permitir que se pueda *detener* el servidor.

Esto es **estrictamente necesario** para que pasen los tests. Simplemente, añadiremos un **nuevo método** a `SimpleREST.java`:

```
public class SimpleREST {
    private Component component;

    public void runServer() { /* ... */ }

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

Cuarto paso: Poner en marcha el tinglado

Para probar lo que acabamos de hacer, basta con *instanciar* nuestra clase desde `Main.java` e *invocar* el método `runServer`.

```
SimpleREST myServer = new SimpleREST();
myServer.runServer();
```

Observa que, cuando lo hagas, verás el mensaje:

```
Starting the internal [HTTP/1.1] server on port 8104
```

...y la ejecución **se mantiene activa** hasta que pulsas el botón de **Re-run**, de **Stop**, o bien sales de IntelliJ IDEA.

¿Pero esto sirve de algo?

Sí.

Lanza la aplicación servidor.

Dirígete a tu navegador web y escribe de nuevo en la barra de direcciones:

```
http://localhost:8104/example
```

¿Ves el resultado?

¡Felicidades! Has implementado tu primer *endpoint* REST usando Java.

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