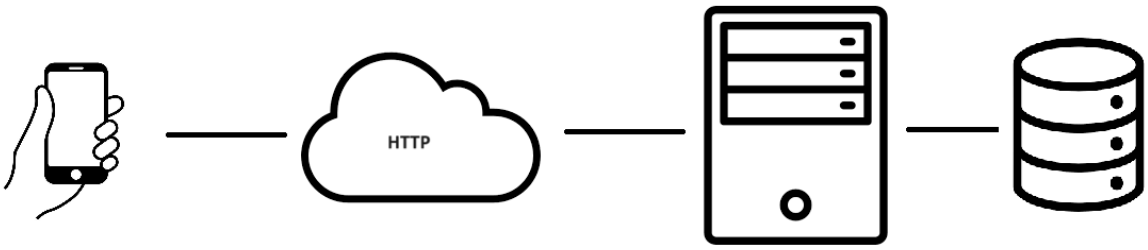
Servicios Web, REST y HTTP

Una de las utilidades más obvias de las aplicaciones, tanto corporativas como App móviles, consiste en alimentarlas con información obtenida de **Web Services** (Servicios Web). Lo habitual es una arquitectura Cliente-Servidor tradicional: un servidor que haga de intermediario de una Base de Datos; y un cliente que se conecta a ese servidor. Si bien un usuario podría operar con su aplicación a nivel local, por detrás, dicha aplicación estaría realizando peticiones al servidor y éste le responde con la información solicitada.

Hemos visto que existen diferentes protocolos para diferentes funcionalidades: FTP, HTTP, SMTP… Nosotros vamos a centrarnos en **HTTP**.



A nivel elemental, este tipo de arquitecturas funcionan de la siguiente forma:

- El cliente necesita un servicio que le ofrece el servidor. El servidor tiene expuestos una serie de servicios que son accesibles por la red.

- El cliente realiza una petición al servidor en forma de mensaje, y lo hace siguiendo las reglas del protocolo que esté usando. Se queda esperando a que el servidor responda.

- El servidor procesa la petición, que usualmente requiere del acceso a una BBDD. Genera un mensaje de respuesta y se la envía de vuelta al cliente.

- Normalmente (pero no siempre), la información intercambiada se hace mediante ficheros en formato xml / json.

- Normalmente, la información intercambiada se cifra de alguna forma. Por ejemplo, usando HTTPS en lugar de HTTP.

Pongamos un ejemplo práctico:

- Un cliente quiere loguearse desde su App del móvil.

- La App cliente solicita al servidor la información del usuario. Le envía los parámetros necesarios (el login).

- El servidor, que es completamente pasivo, procesa la petición. Accede a la base de datos con el login del usuario y recupera sus datos.

- El servidor genera una respuesta empaquetando la información del usuario en un json. Añade también un código de respuesta para indicar si la operación ha sido un éxito o no.

- El cliente compara a la información del servidor y decide si el login & pass es correcto o incorrecto.

Varias tecnologías implicadas en los **Servicios Web**:

Spring Boot es una herramienta para desarrollar aplicaciones web que utiliza el framework **Spring**. En pocas palabras, nos genera automáticamente un programa que puede funcionar como servidor de **Web Services**. Podemos customizarlo y programarlo a nuestro gusto añadiéndole funcionalidades.

Ni que decir tiene que luego podrás tú añadirle más utilidades a tu programa, como Hibernate o JTA para acceder a una Base de Datos.

REST significa *Transferencia de Estado Representacional*. Hablar de REST es hablar de un tipo de Arquitectura en concreto. Actualmente, REST en un sentido muy amplio se refiere a tener un Servidor que es capaz de realizar operaciones sobre una Base de Datos (normalmente), usando **HTTP** para la comunicación y pasando información entre Cliente / Servidor mediante XML, JSON, etc.

Retrofit es un cliente de servidores REST para Android. Permite crear una API de red de forma sencilla que permita recuperar información de un Web Service. Permite, abstraer al programador de las complicaciones del manejo de HTTP, y es capaz de interpretar JSON/XML.

El Protocolo HTTP

HTTP es un protocolo orientado a transacciones que sigue el esquema petición-respuesta entre un Cliente y un Servidor. El cliente realiza una petición enviando un mensaje con cierto formato al servidor, que envía siempre un mensaje de respuesta. Estos mensajes **HTTP** son texto plano.

Programas como el **Postman** nos permiten enviar manualmente y recibir mensajes HTTP enviados a servidores; ideal para testeo.

Peticiones HTTP

En cada una de las peticiones del cliente se incluye la acción requerida por el servidor, la URL del recurso y la versión del HTTP.

Ejemplo de mensaje HTTP GET

```
GET /index.html HTTP/1.1
Host: www.ejemplo.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: es-ES,es;q=0.9
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

En este caso, enviamos al servidor:

- GET: el método HTTP usado para pedir un recurso. En este caso particular, solicita el recurso /index.html (una página web)
- Host: dominio del servidor.
- User-Agent: identifica el navegador o cliente.
- Accept: indica qué tipos de contenido acepta el cliente.
- Connection: keep-alive, mantiene la conexión abierta para más peticiones.

Esta es una petición estándar muy habitual en los navegadores web. Cuando escribimos en el navegador una url como, por ejemplo:

https://www.elorrieta.eus/

Lo que está haciendo por detrás es algo similar al ejemplo que hemos visto: enviar un mensaje HTTP para solicitar la página web de Elorrieta.

Respuestas HTTP

Las respuestas son generadas por el servidor. Incluyen la versión del HTTP, un código de estado, y la propia respuesta del servidor normalmente en el cuerpo del mensaje. HTTP es un protocolo bastante simple.

```
HTTP/1.1 200 OK
Date: Mon, 10 Nov 2025 16:55:00 GMT
Server: Apache/2.4.54 (Ubuntu)
Content-Type: text/html; charset=UTF-8
Content-Length: 137

<!DOCTYPE html>
<html>
<head><title>Ejemplo</title></head>
<body><h1>¡Hola desde el servidor!</h1></body>
</html>
```

En este caso, el servidor responde:

- HTTP/1.1 200 OK: código de estado (200 - éxito).
- Cabeceras como Date, Server, Content-Type y Content-Length.
- El cuerpo del mensaje, en este caso, el HTML de la página solicitada.

Mensajes HTTP

Cada **mensaje** HTTP, ya sea petición o respuesta, está dividido en dos partes:

- Una **cabecera** con metadatos. La cabecera es obligatoria, pero sus campos no tienen por qué serlo.

- Un **cuerpo** del mensaje, opcional. Típicamente tiene los datos que se intercambian entre el Cliente y Servidor. A veces simplemente es necesario responder al cliente con un 'Ok', por lo que no se utiliza el cuerpo del mensaje.

Métodos HTTP

Existen varios **métodos** diferentes que podemos enviar al servidor. Las más conocidas, muy usadas por los Navegadores web, son:

- **GET:** Se usan los GET para solicitar el Servicio especificado, normalmente una página web en el caso de un Navegador web. No suelen tener cuerpo.

- **POST:** Envía datos a un Servicio. Se usa por ejemplo cuando queremos enviar los datos de un formulario de una página web al servidor. Suele tener cuerpo.

Pero no son las únicas acciones. Estas son simplemente las más utilizadas en el caso de la Web. Verás que podemos programar un servidor que use el repertorio completo de **métodos** HTTP como, por ejemplo: DELETE, PUT, HEAD, LOCK…

- **PUT:** Se usa para actualizar un recurso. Suelen tener cuerpo.

- **DELETE**: Se usan para borrar un recurso. A veces tienen cuerpo.

- **HEAD**: Equivale a un GET, pero solo devuelve cabeceras.

- Etc.

Nosotros vamos a usar HTTP para conectarnos a un Servicio Web. Estos servicios Web son a nivel elemental, una serie de métodos java que pueden 'invocarse' desde el cliente. Así es como trabajaremos contra una BBDD remota.

Existe una equivalencia entre los **métodos HTTP** y las **operaciones CRUD** de Base de Datos. Un GET equivale a una SELECT, un POST equivale a una INSERT, un DELETE es un DELETE, etc.

Por defecto, se toma:

HTTP	**SQL**
GET	Select
POST	Insert
PUT	Update
DELETE	Delete

Aunque tienes libertad para definir lo que quieras como quieras, es mala idea saltarse este estándar sin un buen motivo.

Código de estado HTTP

Las **respuestas** de un Servidor generan siempre mediante códigos de estado, independientemente de si hay un cuerpo o no. Estos códigos son números que indican lo que ha pasado en el servidor al procesar la petición.

- **Códigos con formato 1xx**: Respuestas informativas. Indica que la petición ha sido recibida y se está procesando.
- **Códigos con formato 2xx**: Respuestas correctas. Indica que la petición ha sido procesada correctamente.
- **Códigos con formato 3xx**: Respuestas de redirección. Indica que el cliente necesita realizar más acciones para finalizar la petición.
- **Códigos con formato 4xx**: Errores causados por el cliente. Indica que ha habido un error en la petición porque el cliente ha hecho algo mal.
- **Códigos con formato 5xx**: Errores causados por el servidor. Indica que ha habido un error en la petición a causa de un fallo en el servidor.

Corresponde a los programadores del lado Servidor decidir cuándo se envía cada código. De la misma forma, corresponde a los programadores del lado Cliente decidir cómo se trata cada uno de esos códigos HTTP.

Varios de los códigos de respuesta más comunes son:

100 Continue. Todo hasta ahora está bien y el cliente debe continuar con la solicitud o ignorarla

200 OK. La solicitud ha tenido éxito. Si era por ejemplo un:

GET: El recurso se ha obtenido y se transmite en el cuerpo del mensaje

PUT o POST: El recurso que describe el resultado de la acción se transmite en el cuerpo del mensaje.

201 Created. Se ha creado un nuevo recurso. Respuesta típica de PUT

301 Moved Permanently. La URI del recurso ha cambiado. La nueva URI se devolverá en la respuesta si se conoce.

400 Bad Request. El server no entiende, sintaxis inválida

401 Unauthorized. Tienes que autenticarte.

403 Forbidden. No tienes permiso de acceso

404 Not Found. No podemos encontrar el recurso.

Usando HTTP [Java]

Hasta la **versión 1.8 de Java**, la clase HttpURLConnection era la base de la programación de aplicaciones capaces de acceder a servicios web.

Los pasos para crear una petición HTTP sencilla son:

1) Crear la URL del recurso objetivo
2) Abrir la conexión
3) Configurar la conexión
    a. El método HTTP que usamos: Get, Post, Put, Delete
    b. Tipo de contenido
    c. Sistema de codificación
    d. Agente de usuario a usar

4) Obtención y evaluación del **código de estado**.
    a. Si es 200, obtener InputStream para leer y el OutputStream para escribir. Desconectar.
    b. Si no es 200, tratar el código de respuesta (que es un error).

Tienes un ejemplo en la clase **AccesoRAE** En este ejemplo, se solicita a un servidor web que nos envíe una página web en base a su URL.

A partir de **Java 11**, el paquete http ofrece una forma más potente, sencilla y actualizada de realizar peticiones HTTP.

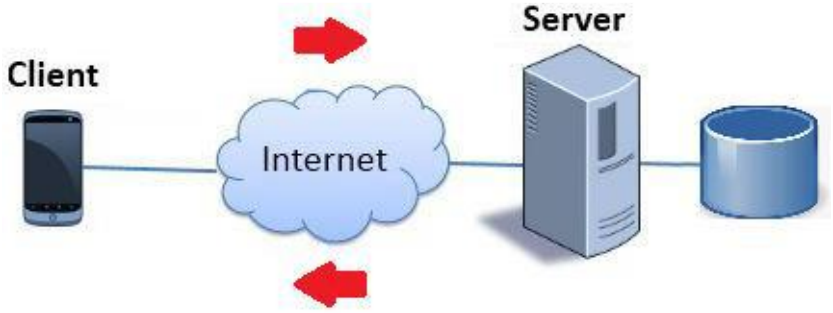
Los pasos para crear una petición HTTP son:

1) Crear un HttpClient
2) Crear un HttpRequest indicando la URI y otros parámetros de cabecera de la request
3) Realiza la petición mediante send () y obtener la respuesta.
4) Procesar la respuesta.

Tienes un ejemplo en la clase **AccesoRAEHttp**

Seamos prácticos: HTTP y REST

REST significa *Transferencia de Estado Representacional*. Hablar de REST es hablar de un tipo de Arquitectura en concreto. Actualmente, REST en un sentido muy amplio se refiere a tener un Servidor que es capaz de realizar operaciones sobre una Base de Datos (normalmente), usando **HTTP** para la comunicación y pasando información entre Cliente / Servidor mediante XML, JSON, etc.



Una explicación simple del funcionamiento de una Api REST sería la siguiente:

- El **Servidor** tiene una serie de métodos expuestos al exterior. Esto significa que un **Cliente** puede ejecutar esos métodos como si fuese su propio código.

- Estos métodos están **mapeados**, de forma que cada uno se corresponde con una request o petición HTTP diferente.

- Estas request toman la forma de URL, y pueden ser Get, Post, etc. Por ej.:

  http://localhost:8080/PruebasRestFull/rest/hello/javatpoint

- Cuando un **Cliente** lanza esta Request (petición HTTP) el método asociado en el lado del servidor se ejecuta, genera un resultado y lo devuelve al cliente.

- Los mensajes que viajan por Internet en el cuerpo de una petición son texto plano (XML, JSON), por tanto, será necesario hacer conversiones de datos.

- Las **respuestas** del Servidor son **códigos de respuesta HTTP**.

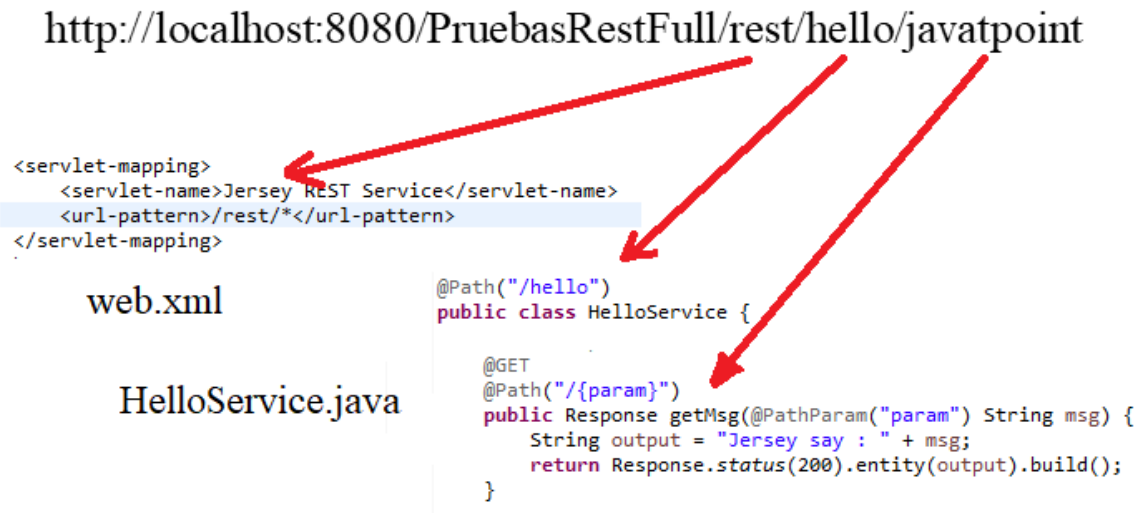
Un ejemplo de una **request** (petición) podría ser 'traducida' así:

http://localhost:8080/PruebasRestFull/rest/hello/javatpoint

Lo desglosamos en:

- **Http://** → Protocolo de comunicación que se está usando.
- **Localhost:8080** → La IP y el Puerto del Servidor destino.
- **MiProyecto** → El proyecto, el **recurso**, el nombre del servicio que queremos invocar. Esto le permite al **Servidor** diferenciar el recurso al que nos estamos refiriendo. Los Servidores normalmente tienen docenas de estos servicios.
- **Rest** → Mapeo del Servlet. Reconoce lo que viene detrás como llamadas a métodos REST
- **Hello** → El path de la clase. Esto es configurable en cada proyecto, clase o método mediante el uso de Anotaciones.
- **Javapoint** → Es un atributo que le pasamos al Server, en este caso un String.
- NO puede verse simplemente con la URL, pero esta Request se corresponde a un **GET**. Lanzamos una petición sobre un servicio que ejecuta una Select sobre Base de Datos, y nos retorna el resultado.

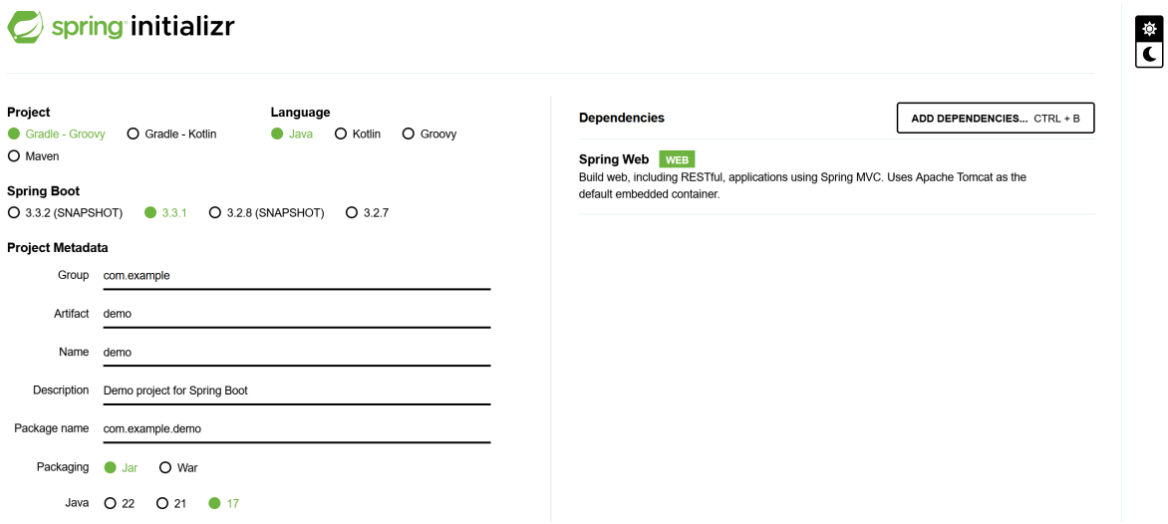
Por tanto, esta URL estaría ejecutando un **Servlet**:



Springboot

Como ya hemos mencionado, **Spring Boot** es una herramienta que nos permite desarrollar Apis Rest rápidamente. Vamos a montar un pequeño servidor con **Spring Boot** que exponga unos pocos servicios a la red. Probaremos estos servicios desde **Postman**, y después prepararemos otros tantos para ser accesibles desde una App utilizando **Retrofit (o REST si no es una app móvil)**.

Para crear nuestro proyecto con Springboot, usaremos el **Spring Initializr**. Accedemos a la página **https://start.spring.io/** donde configuraremos los parámetros que tendrá nuestro servidor. Una vez hecho, esta página nos permitirá descargar el código fuente del servidor con la configuración indicada. Es una forma sencilla y cómoda de empezar un proyecto, en comparación a hacerlo de forma manual.



En nuestro caso, indicaremos:

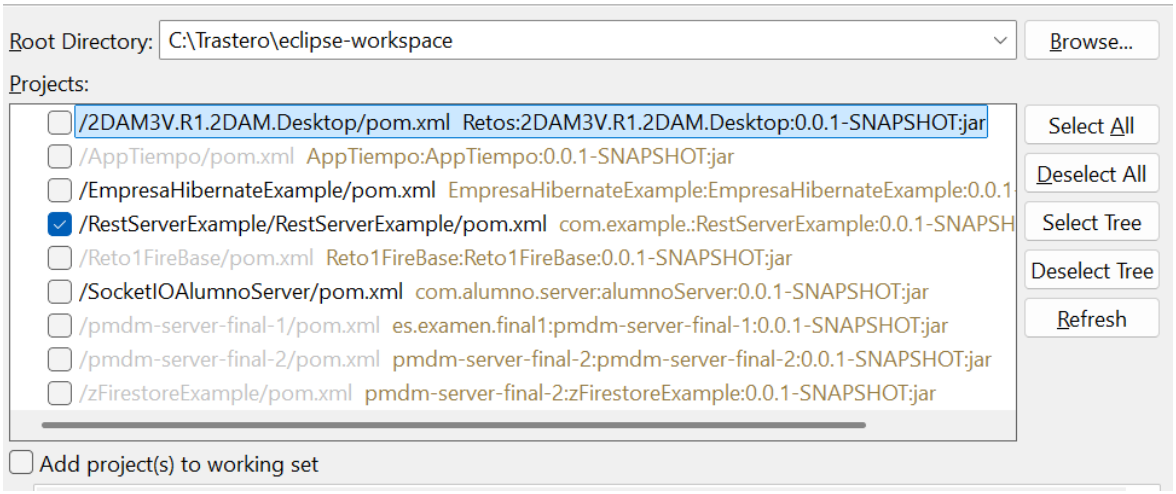
1. **Project:** Maven Project
2. **Language:** Java
3. **Spring Boot:** La versión estable más reciente
4. **Artifact & Name:** RestServerExample (para nuestro ejemplo)
5. **Packaging:** Jar
6. **Java:** 17 o superior
7. Añade la dependencia: **Spring Web**

No te olvides de indicar la dependencia de Spring Web, que incluye las librerías necesarias para los Web Services. Si no lo haces, tendrás que repetir el proceso.

Al darle a **Generate**, se nos descargará un RestServerExample.zip con el código del proyecto.

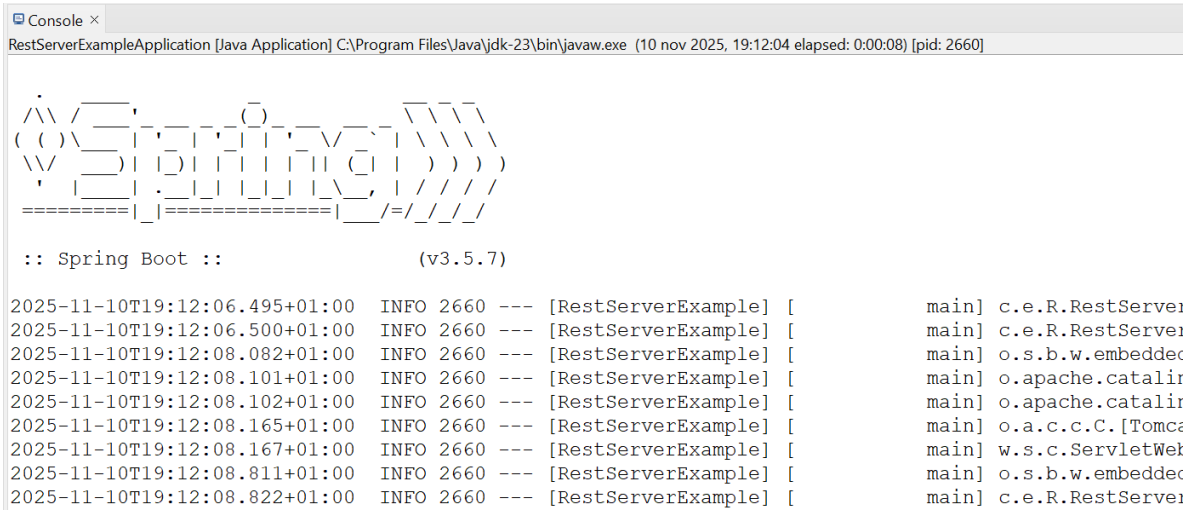
Desplegando el proyecto en Eclipse

Para hacerlo desplegar el proyecto en Eclipse, pega el fichero RestServerExample.zip en la raíz de tu workspace, y descomprímelo. A continuación, vamos a **File / Import / Existing Maven Projects**. Allí, le ponemos la ruta de la raíz del workspace y seleccionamos el pom.xml del proyecto que queremos importar:



Una vez se cierre la ventana, seleccionas el fichero pom.xml y le das botón derecho / Maven / Update Project.

Puedes ejecutar el proyecto desde la clase RestServerExampleApplication.java. Verás que la consola muestra un montón de texto, aunque en realidad, el programa no hará nada. Esto es porque el servidor estará en espera de que le lleguen peticiones HTTP. Y para eso, hay que definir primero los **endpoints**.



Montando un ejemplo sencillo

Un **endpoint** es **una dirección (URL) concreta** dentro de un servicio web REST donde se puede acceder o manipular un recurso. En otras palabras, es una función que está expuesta a la red, y es accesible a través de la URL mediante el protocolo HTTP.

Para poder exponer estos **métodos** tienen que estar **correctamente anotados**.

Crea el paquete controllers y el paquete entities. Vamos a utilizarlos como si fuesen una Base de Datos real. Obviamente, aquí iría todo el código necesario para acceder a dicha BBDD.

Generamos primero **Alumno**, una entidad con unos atributos básicos.

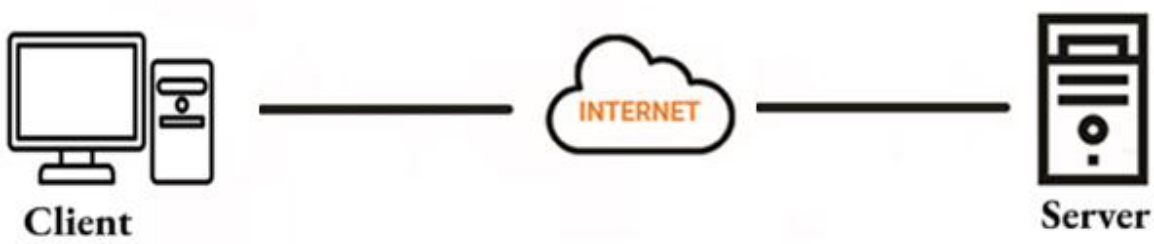
```
public class Alumno implements Serializable {

    @Serial
    private static final long serialVersionUID = -3126598149146249922L;

    private int id = 0;
    private String nombre = null;
    private String apellido = null;
```

Esta entidad NO tiene nada que ver con la Entidad de Base de Datos; aunque su utilidad es similar. Va a ser utilizada para enviar en el Mensaje HTTP la información relativa a un Alumno.

Es decir: al hacer una petición **HTTP**, lo que va a viajar con ella es mi Entidad.



Por cierto, aclarar que, en el caso de que reutilices las Entidades para la comunicación HTTP (Capa Vista-Controlador), deberías usar otras diferentes para las Bases de Datos (Capa Controlador-Modelo). Estrictamente hablando. Aunque sean iguales las unas que las otras.

Ahora que ya tenemos la Entidad Alumno que va a ir embebida con los mensajes HTTP, toca preparar al servidor para atender las peticiones de los clientes. Eso se hace mediante una clase **Controlador** (que es un nombre inventado para los ejemplos solamente).

Acudimos a la carpeta controllers y generamos la clase **AlumnoController**. Esta clase va a **agrupar todos los servicios** que tengan que ver con Alumno. Puedes hacerlo como prefieras en realidad, pero es mejor ser ordenado.

```
@RestController
public class AlumnoController {

    public AlumnoController () {
        super();
    }
```

No olvides anotar la clase como un Controlador REST.

Dado que nuestra App va a querer consultar y añadir alumnos a la Base de Datos del servidor, vamos a emular que tenemos una llamada **FakeDataBase**. Esta clase es un Singleton que tiene un sencillo ArrayList de alumnos, con unos cuantos métodos auto explicativos. Obviamente, en un proyecto real se accedería a una Base de Datos mediante Hibernate, JTA, etc. pero para nosotros nos basta con esto.

```
public class FakeDataBase {

    private static final FakeDataBase instance = null;

    private List<Alumno> alumnos = null;

    private FakeDataBase() {
        alumnos = new ArrayList<Alumno>();
        alumnos.add(defaultAlumno());
    }

    public static FakeDataBase getInstance() {
        return instance == null ? new FakeDataBase() : instance;
    }

    public void reset() {
        alumnos.clear();
        alumnos.add(defaultAlumno());
    }

    private Alumno defaultAlumno() {
        return new Alumno("Juan", "Torres");
    }
```

Vamos a añadir un **endpoint** (que te recuerdo que es una URL). Los endpoints están relacionados con los cuatro tipos de operaciones básicas HTTP: Get, Post, Put y Delete. Que a su vez estarán relacionadas con las operaciones SQL de Select, Insert, Update y Delete. Te recuerdo La tabla de antes:

HTTP	**SQL**
GET	Select
POST	Insert
PUT	Update
DELETE	Delete

Por tanto, si queremos hacer que un método nos retorne todos los alumnos (un Select * from t_alumnos) tendríamos que añadir a **AlumnoController** un método similar a este:

```
public List <Alumno> getAllPotatoes (){
    List <Alumno> ret = null;
    FakeDataBase fake = FakeDataBase.getInstance();
    ret = fake.getAllAlumnos();
    return ret;
}
```

Sólo nos quedan los últimos retoques. Hay que **anotar** el método endpoint:

```
@GetMapping ("/alumnos")
public List <Alumno> getAllPotatoes (){
    List <Alumno> ret = null;
    FakeDataBase fake = FakeDataBase.getInstance();
    ret = fake.getAllAlumnos();
    return ret;
}
```

¿Qué significa todo esto?

- **GetMapping** es la anotación que indica que el método es para una operación Get de HTTP. Entre paréntesis se indica el **path** que referenciará al método.

- El nombre del método es irrelevante para el funcionamiento del endpoint.

- Llamar a este método expuesto mediante HTTP retorna una Lista de Alumnos en el cuerpo de la respuesta del Get.

Por tanto, esta anotación sirve para **montar la URL** que hace referencia a este método. En pocas palabras, si escribes esa URL en un navegador web, por ejemplo, en lugar de cargar una página se llamará a este método (puede que no veas nada si lo haces).

La URL sería: http://localhost:8080/alumnos

No nos quedemos solamente con un **Get**. Hagamos también un **Post**. Implementemos un endpoint para insertar un alumno en la BBDD.

```
@PostMapping ("alumno/new")
public String addAlumno (@RequestBody Alumno alumno) {
    FakeDataBase fake = FakeDataBase.getInstance();
    fake.addAlumno(alumno);
    return "Alumno insertado";
}
```

¿Qué significa todo esto?

- **PostMapping** es la anotación que indica que el método es para una operación Post de HTTP. Entre paréntesis está el **path** que referenciará al método.
- El nombre del método es irrelevante para el funcionamiento del endpoint.
- En este caso, retornamos un String, pero el Post de Http no tiene cuerpo de respuesta, por lo tanto, no es habitual que un Post devuelva algo.

La URL sería: http://localhost:8080/alumno/new

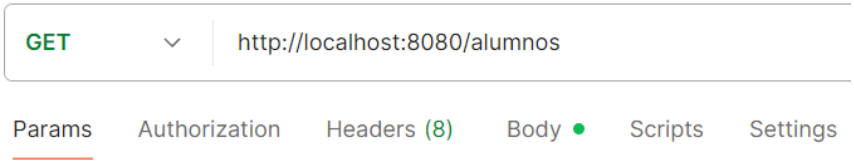
Al conjunto de métodos expuestos que permiten acceder contra una Base de Datos se le llama **Api REST**.

Probando desde Postman

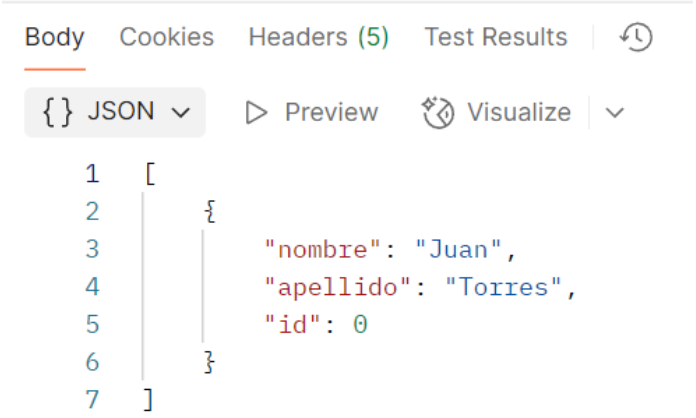
El programa **Postman** nos permite Postman probar APIs REST sin necesidad de escribir código. Funciona como un "cliente" donde puedes enviar peticiones HTTP fácilmente (GET, POST, PUT, DELETE…) generadas al vuelo, sin tener que ejecutar un cliente real, ver la respuesta y depurar problemas.

Vamos a probar los dos métodos de nuestra Api.

Probamos el **Get**: http://localhost:8080/alumnos



El resultado del envío de la petición es:

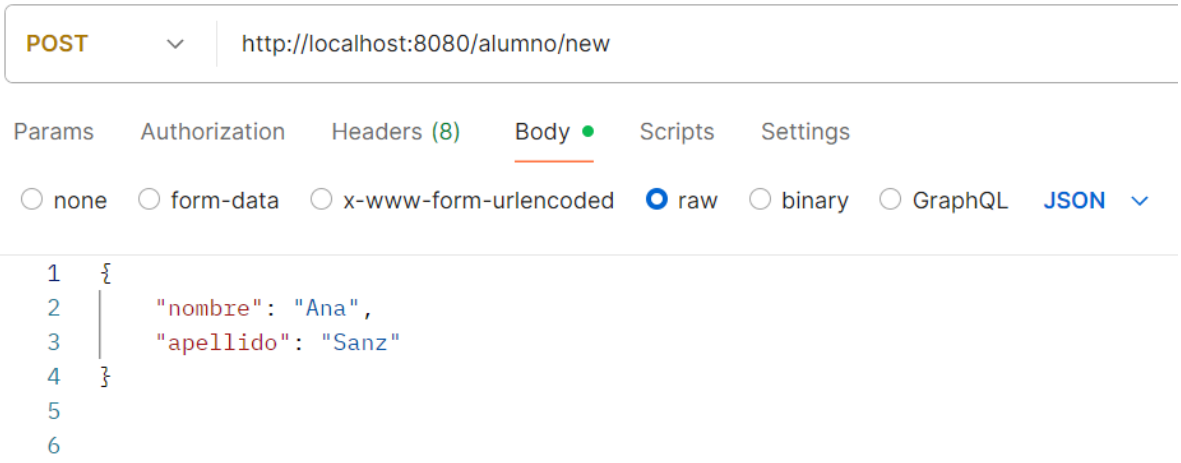


Efectivamente: es un JSON con la información de nuestro alumno en BBDD. Puedes configurar para que sea un JSON o un XML, pero es mejor el primero.

Este JSON viaja en el cuerpo de la respuesta del HTTP – GET. Pero no tiene por qué ser un JSON: puedes incluir cualquier cosa aquí.

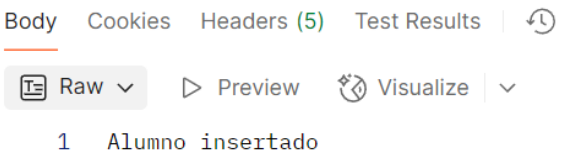
Veamos ahora el **POST**: http://localhost:8080/alumno/new

En este caso, tenemos que enviarle un objeto Alumno, pero en formato JSON. Dado que un Alumno tiene de atributos nombre y apellido…



Nótese que la id no está puesta en el constructor de Alumno.

El resultado del envío de la petición es:



Los **errores** pueden verse en el Postman mediante **códigos de estado HTTP**.