Unión Europea

Fondo Social Europeo

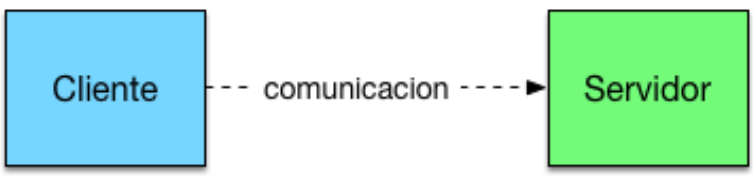
El FSE invierte en tu futuro

Fecha	Versión	Descripción
05/12/2022	1.0.0	Versión inicial.

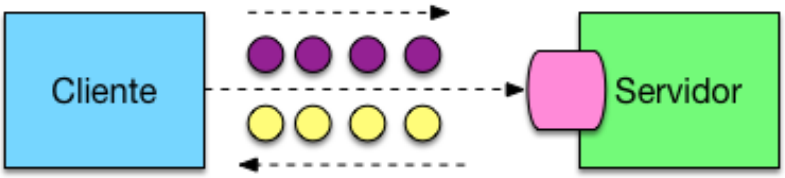
Unidad 7 - Servicios REST

1. ¿Qué es REST?

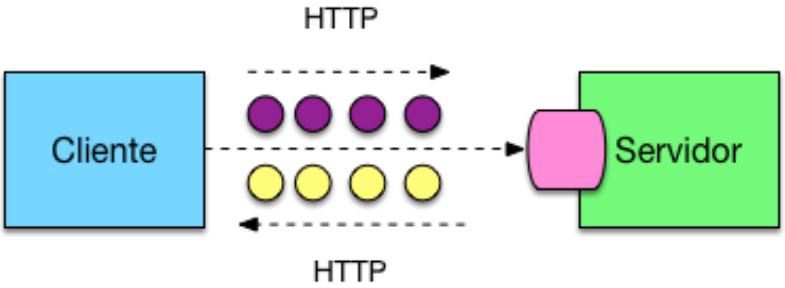
Esta pregunta es una de las más habituales en nuestros días. Para algunas personas REST es una arquitectura, para otras es un patrón de diseño, para otras un API. ¿Que es REST exactamente? . REST o *Representational State Transfer* es un estilo de Arquitectura a la hora de realizar una comunicación entre cliente y servidor.



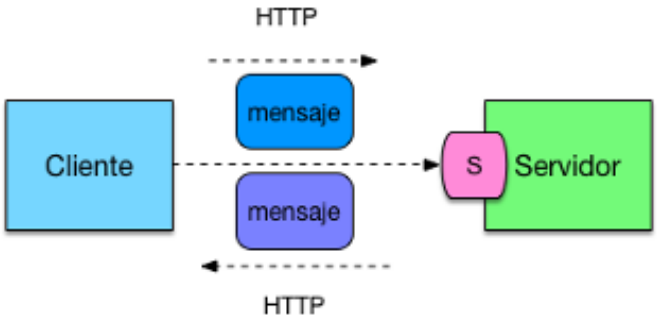
Vamos a intentar explicarlo esto paso a paso. Habitualmente cuando nosotros realizamos una comunicación cliente servidor accedemos al servidor en un punto de acceso, le enviamos una información y recibimos un resultado.



Ahora bien, hay muchas formas de realizar esta operación. ¿Cuál es la más correcta? Esa es una buena pregunta. Hoy por hoy una de las necesidades más claras es que esa comunicación sea abierta y podemos acceder desde cualquier sitio. Así pues, estamos hablando de una comunicación HTTP.



Una vez tenemos claro el protocolo de comunicación el siguiente paso es decidir que tipología de mensajes enviamos. Como punto de partida podemos mandar a un servicio un mensaje en formato XML o JSON. El servicio lo recepcionará y nos devolverá una respuesta.



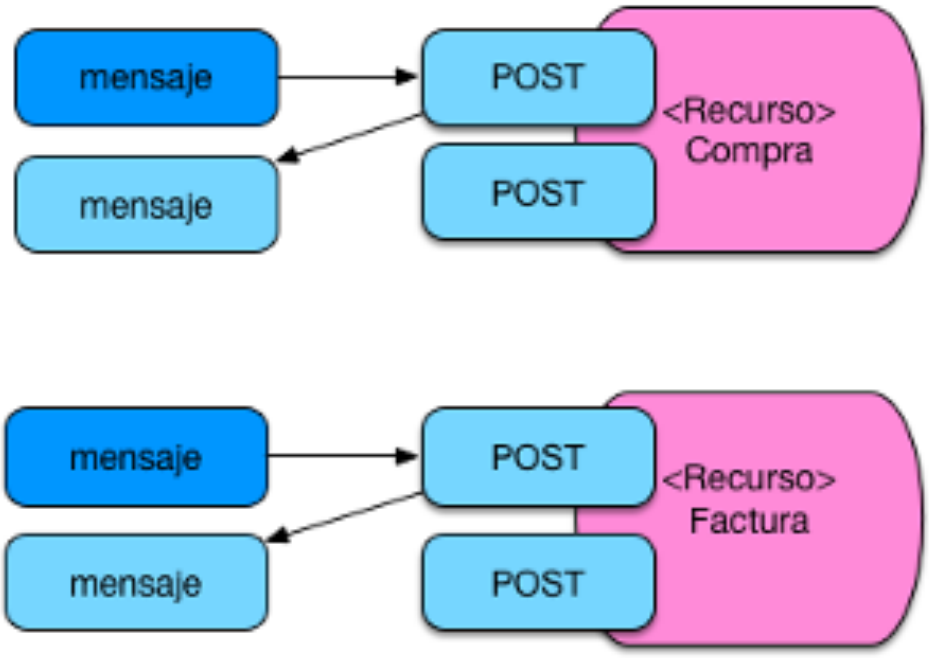
Esto es lo que habitualmente en Arquitecturas REST se denomina el nivel 0. No tenemos ningún tipo de organización. Es el caos.

REST nació por la necesidad de simplificar la creación de Web Services utilizando el protocolo HTTP como base. Este es una forma "ligera" y rápida de desarrollar y consumir Web Services. Debido a que el protocolo HTTP es utilizado prácticamente en todos lados donde utilicemos la Web, es posible reutilizar este mecanismo de comunicación como la base para la transmisión de Servicios Web.

1.1. Nivel 1: Recursos

El siguiente paso es lo que se denomina nivel 1, en vez de tener servicios con métodos diversos declaramos Recursos.

Un Recurso hace referencia a un concepto importante de nuestro negocio (Facturas ,Cursos , Compras etc). Este estilo permite un primer nivel de organización permitiendo acceder a cada uno de los recursos de forma independiente, favoreciendo la reutilización y aumentando la flexibilidad.



1.2. Nivel 2: HTTP Verbs

Hasta este momento para realizar las peticiones se usa GET o POST indistintamente. En el nivel 2 las operaciones pasan a ser categorizadas de forma más estricta. Dependiendo de cada tipo de operación se utilizará un método diferente de envío.

- **GET:** Es utilizado únicamente para **consultar información** al servidor, muy parecidos a realizar un SELECT a la base de datos. No soporta el envío del payload (consiste en la información que se adjunta al llamado Web Services, como información necesaria para la acción que estamos realizando).
- **POST:** Es utilizado para solicitar la **creación de un nuevo registro**, es decir, algo que no existía previamente, es decir, es equivalente a realizar un INSERT en la base de datos. Soporta el envío del payload.
- **PUT**: Se utiliza para **actualizar por completo un registro existente**, es decir, es parecido a realizar un UPDATE a la base de datos. Soporta el envío del payload.
- **PATCH**: Este método es similar al método PUT, pues permite actualizar un registro existente, sin embargo, este se utiliza cuando **actualizar solo un fragmento del registro** y no en su totalidad, es equivalente a realizar un UPDATE a la base de datos. Soporta el envío del payload
- **DELETE**: Este método se utiliza para **eliminar un registro existente**, es similar a DELETE a la base de datos. No soporta el envío del payload.
- **HEAD**: Este método se utilizar para **obtener información sobre un determinado recurso** sin retornar el registro. Este método se utiliza a menudo para probar la validez de los enlaces de hipertexto, la accesibilidad y las modificaciones recientes.

Este es el nivel en el que hoy en día se encuentran muchas de las Arquitecturas REST.

La forma de crear un Web Services utilizando REST es a través de recursos (resources). Cada recurso tiene asociado una URI, y cada URI representa a su vez una operación del Web Service.

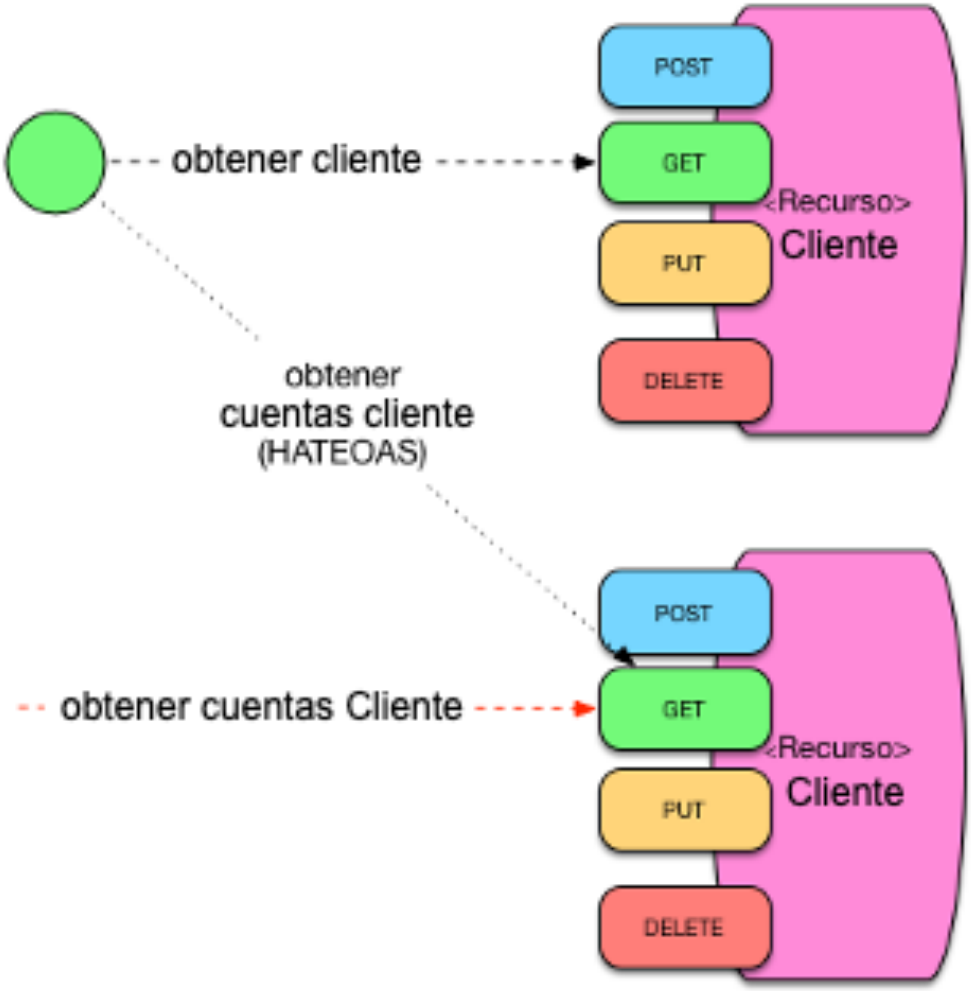
- `/clientes` : Es una URI que representa todas las entidades de tipo Cliente de nuestro sistema.
- `/clientes/{id}` - Esta URI representa una orden en particular. A partir de esta URI, podremos leer (*read*), actualizar (*update*) y eliminar (*delete*) objetos de tipo Cliente.

Hasta aquí los métodos más utilizados en la construcción de servicios REST con Spring Boot, sin embargo, existen algunos métodos más que son interesantes conocer, pues nos los encontraremos al momento de depurar o analizar el tráfico de red.

- **CONNECT**: Se utiliza para establecer una comunicación bidireccional con el servidor. En la práctica no es necesario ejecutarlo, si no el mismo API de HTTP se encarga de ejecutarlo para establecer la comunicación previo a lanzar alguna solicitud al servidor.
- **OPTIONS**: Este método es utilizado para describir las opciones de comunicación para el recurso de destino. Es muy utilizado con CORS (Cross-Origin Resource Sharing) para validar si el servidor acepta peticiones de diferentes origines. Este lo utilizaremos para configurar nuestros servicios.

1.3. Nivel 3: HATEOAS

Todavía nos queda un nivel más que es el que se denomina *HATEOAS* (*Hypertext As The Engine Of Application State*). ¿Para qué sirve este nivel?. Vamos a poner un ejemplo mediante clientes y cuentas: Supongamos que queremos acceder a un recurso Cliente vía REST . Si tenemos una Arquitectura a nivel 2 primero accederemos a ese recurso utilizando GET. En segundo lugar deberemos acceder al recurso de Cuentas para añadir al cliente las cuentas (línea roja).



Esto implica que el cliente que accede a los servicios REST asume un acoplamiento muy alto, debe conocer la URI del Cliente y la del Cuentas. Sin embargo, si el recurso de Cliente contiene un link al recurso de Cuentas esto no hará falta. Podríamos tener una estructura JSON como la siguiente:

```
[
  {
    "id" : 1,
    "nif" : "44999594K",
    "nombre" : "Keiko",
    "apellidos" : "Meadows",
    "claveseguridad" : "1732",
    "email" : "meadowskeiko6933@icloud.net",
    "fechanacimiento" : "2023-01-03"
        "links": [
            {
                "rel": "self",
                "href": "http://localhost:8888/clientes/1"
            },
            {
                "rel": "listaDirecciones",
                "href": "http://localhost:8888/clientes/1/direcciones"
            },
```

```
            {
                "rel": "listaCuentas",
                "href": "http://localhost:9090/clientes/3/cuentas"
            }
        ]
    }
]
```

Podremos acceder directamente al recurso de la lista de cuentas (o direcciones) utilizando las propiedades del Alumno. Esto es HATEOAS, lo que permite aumentar la flexibilidad y reducIR el acoplamiento. Construir arquitecturas sobre estilo REST no es sencillo y hay que ir paso a paso.

Esto implicaria modificar los DTO's para que, en lugar de tener una lista de direcciones o cuentas tendremos un link al recurso que se traera cada lista.

2. Métodos HTTP

Con estos métodos debemos tomar en cuenta dos características muy importantes. Los métodos seguros e idempotentes.

- Los métodos seguros (no modifican es estado del sistema) son aquellos que no hace otra tarea que recuperar información, por ejemplo, los métodos `GET` y `HEAD`.
- Los métodos indempotentes son aquellos que siempre se obtiene el mismo resultado, sin importar cuantas veces se realice cierta operación. Métodos que son idempotentes son: `GET`, `HEAD`, `PUT` y `DELETE`.
- El resto de los métodos no son ni seguros ni idempotentes.

Aunque a nivel de programación es posible realizar más de una operación al momento de enviar una petición, por ejemplo, actualizar y eliminar, esto NO debería programarse de esta manera, ya que rompe con la idea básica de los REST Web Services.

Una gran ventaja de que las operaciones REST sean sobre el protocolo HTTP es que los administradores de servidores, redes y encargados de seguridad de las mismas, saben cómo configurar este tipo de tráfico, por lo que no es algo nuevo para ellos.

3. Request Headers (Cabeceras de Petición)

Las cabeceras de las peticiones permiten obtener metadatos de las peticiones HTTP, como pueden ser:

- El método HTTP utilizado en la petición (GET, POST, etc).
- La IP del equipo que realizo la petición (192.168.1.135).
- El dominio del equipo que realizo la petición (www.midominio.com)
- El recurso solicitado (http://midominio.com/recurso)
- El navegador que se utilizó en la petición (Mozilla, Internet Explorer, etc).

- Etc.

Al enviar y recibir mensajes con REST Web Services, es necesario tener conocimiento de códigos de estado. Algunos de los códigos de estado más utilizados son:

- 200 (Ok): Significa que la respuesta fue correcta, este el código de estado por default.
- 204 (Sin Contenido): El navegador continúa desplegando el documento previo.
- 301 (Movido Permanentemente): El documento solicitado ha cambiado de ubicación, y posiblemente se indica la nueva ruta, en ese caso el navegador se redirecciona a la nueva pagina de manera automática.
- 302 (Encontrado): El documento se ha movido temporalmente, y el navegador se mueve al nuevo url de manera automática.
- 401 (Sin autorización): No se tiene permiso para visualizar el contenido solicitado, debido a que se trató de acceder a un recurso protegido con contraseña sin la autorización respectiva.
- 404 (No encontrado): El recurso solicitado no se encuentra alojado en el servidor Web.
- 500 (Error Interno del Servidor Web): El servidor web lanzó una excepción irrecuperable, y por lo tanto no se puede continuar procesando la petición.

Para una lista completa de los códigos de estado del protocolo HTTP se puede consultar el siguiente link:

http://en.wikipedia.org/wiki/List_of_HTTP_status_codes

4. Cabeceras de Respuesta y Tipos MIME

Las cabeceras de respuesta se utilizan para indicar al navegador Web como debe comportarse ante una respuesta por parte del servidor Web.

Un ejemplo común es generar hojas de Excel, PDF's, Audio, Video, etc, en lugar de solamente responder con texto.

Para indicar el tipo de respuesta se utilizan los tipos MIME (*Multipurpose Internet Mail Extensions*).

Los tipos MIME son un conjunto de especificaciones con el objetivo de intercambiar archivos a través de internet como puede ser texto, audio, video, entre otros tipos.

Los tipos MIME (*Multipurpose Internet Mail Extensions*) son el estándar de internet para definir el tipo de información que recibirá el cliente (navegador Web) al realizar una petición al servidor Web.

Los REST Web Services pueden devolver distintos tipos de contenido para un mismo recurso, por ejemplo:

- *application/xml* -> Mensaje XML
- *application/json* -> Mensaje JSON
- *text/html* -> Salida HTML
- *text/plain* -> Salida en texto plano
- *application/octet-stream* -> Datos binarios

Lo anterior son los tipos de recursos que más se utilizan para el envió de información en REST. Lo más común es utilizar XML, sin embargo no está limitado a este tipo de información, ya que si estamos utilizando un cliente con Jquery y AJAX, es común que utilicemos la anotación de JSON en lugar de XML.

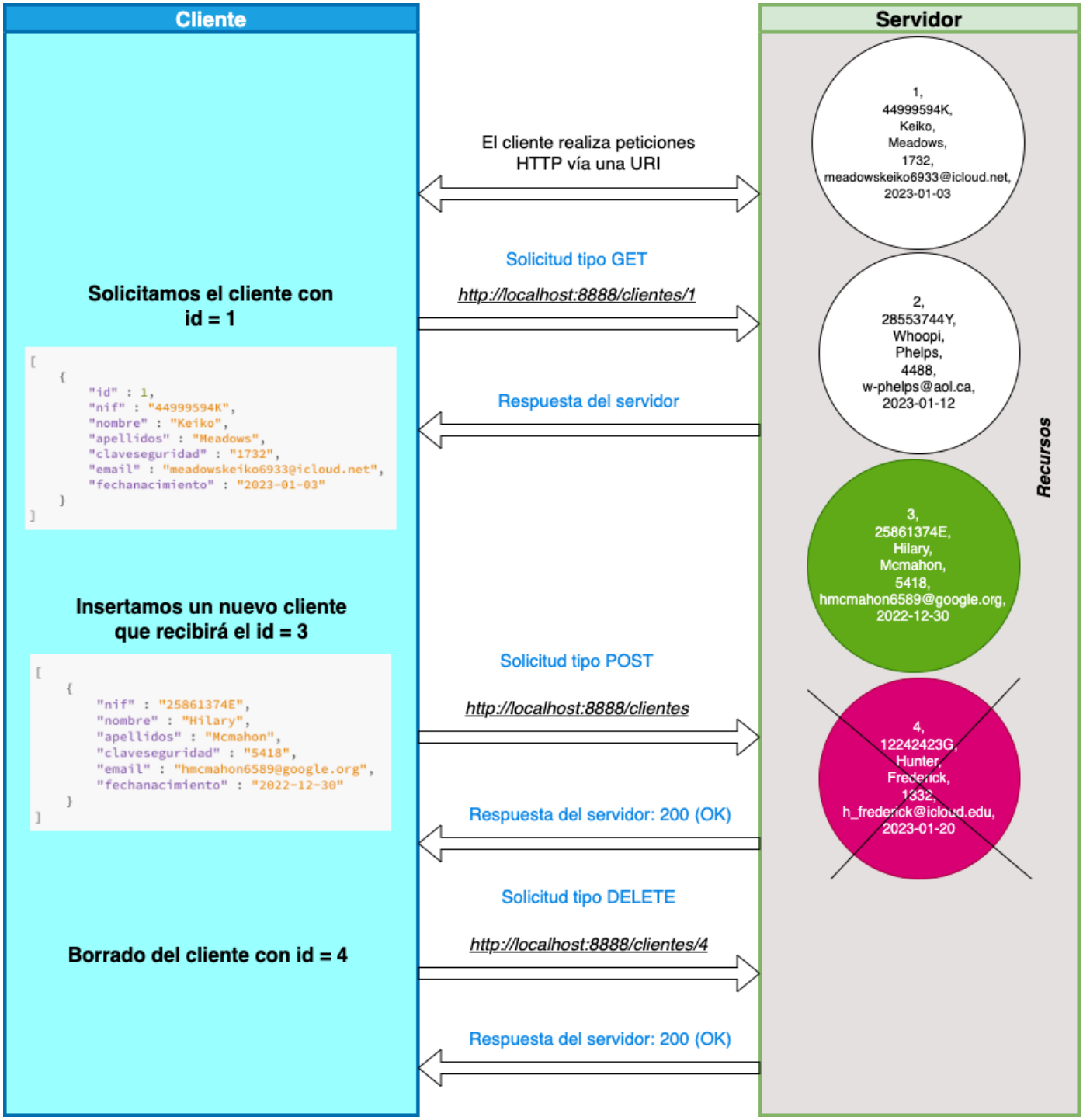
Un listado más completo de tipos MIME se puede consultar el siguiente link:

http://www.freeformatter.com/mime-types-list.html

5. Representaciones con REST y diseño

Cuando el cliente envía una solicitud a través de una API de RESTful, esta transfiere una representación del estado del recurso requerido a quien lo haya solicitado o al extremo. Esto es que los clientes interactúan con un servicio mediante el intercambio de *Representaciones* de recursos, donde estas representaciones pueden venir en formato JSON o XML como formato de intercambio.

Como podemos observar en la figura, REST utiliza el concepto de Representaciones, y cada representación apunta a un Recurso(s) del lado del Servidor Java.




```
[
    {
        "id" : 1,
        "nif" : "44999594K",
        "nombre" : "Keiko",
        "apellidos" : "Meadows",
        "claveseguridad" : "1732",
        "email" : "meadowskeiko6933@icloud.net",
        "fechanacimiento" : "2023-01-03"
    }
]
```

```
[
    {
        "nif" : "25861374E",
        "nombre" : "Hilary",
        "apellidos" : "Mcmahon",
        "claveseguridad" : "5418",
        "email" : "hmcmahon6589@google.org",
        "fechanacimiento" : "2022-12-30"
    }
]
```

Por lo que, a la hora de diseñar un API Rest los principios más importantes a seguir son:

- Las API de REST se diseñan en torno a *recursos*, que son cualquier tipo de objeto, dato o servicio al que puede acceder el cliente.
- Un recurso tiene un *identificador*, que es un URI que identifica de forma única ese recurso. Por ejemplo, el URI de un pedido de cliente en particular podría ser:

Por ejemplo, para recuperar el objeto cliente con id = 5, podemos utilizar el siguiente URI:

http://localhost:8888/clientes/5

Esto regresará la representación del objeto en el tipo MIME solicitado. Por ejemplo, si se solicitó una representación en XML, la respuesta debería ser:

```xml
<Personas>
  <Persona>
    <id>5</id>
    <nif>51724739D</nif>
    <nombre>Diana</nombre>
    <apellidos>Dickerson</apellidos>
    <claveseguridad>5053</claveseguridad>
    <email>d-dickerson5965@aol.net</email>
    <fechanacimiento>2023-01-05</fechanacimiento>
  </Persona>
</Personas>
```

Si es JSON la representación vendrá de la siguiente forma:

```json
[
  {
    "id" : 5,
    "nif" : "51724739D",
    "nombre" : "Diana",
    "apellidos" : "Dickerson",
    "claveseguridad" : "5053",
    "email" : "d-dickerson5965@aol.net",
    "fechanacimiento" : "2023-01-05"
  }
]
```

De manera similar, podemos tener las operaciones básicas para agregar, modificar y eliminar recursos del lado del servidor. Por ejemplo, las siguientes URL son ejemplos para cada una de las acciones mencionadas:

- **Agregar**: *POST /clientes* - Solicita agregar un nuevo recurso. Los datos se especifican en el documento JSON ó XML a enviar. Ej.

```json
[
  {
    "nif" : "65487634T",
    "nombre" : "León",
    "apellidos" : "Sánchez",
    "claveseguridad" : "6478",
    "email" : "leonsanchez3456@aol.net",
    "fechanacimiento" : "1976-02-07"
  }
]
```

- **Modificar**: *PUT /clientes/123* – Solicita modificar el recurso 123 con un JSON ó XML respectivo.
- **Eliminar**: *DELETE /clientes/13* – Solicita eliminar el recurso 13

6. Anotaciones con Spring Boot

Crear una clase para exponer un método como un servicio REST en Spring Boot es muy simple. A continuación mencionaremos algunas de las anotaciones más utilizadas en Spring Boot, sin embargo, existen más anotaciones dependiendo del requerimiento a cubrir.

- `@RestController`: Esta anotación debe aparecer al inicio de la clase e indica que la clase se comportará como un servicio web que devuelve, por defecto, JSON. Con esto Spring Boot preparará todas las dependencias y comportamientos necesarios para que podamos crear las acciones (que a partir de ahora recibirán el nombre de **endpoints**) en este controlador. Por ejemplo:

```
package com.example.demo.web.webservice;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class ClienteRestController {

  private static final Logger log =
LoggerFactory.getLogger(ClienteRestController.class);

}
```

- `@RequestMapping`: permite asignar solicitudes web a clases que manejen la petición o a métodos específicos. Cuando `@RequestMapping` se usa en el nivel de clase, crea un URI base para el que se usará el controlador. A nivel de clase queda de la siguiente forma:

```
package com.example.demo.web.webservice;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/ws/clientes")
public class ClienteRestController {

  private static final Logger log =
LoggerFactory.getLogger(ClienteRestController.class);

}
```

Cuando esta anotación se utiliza en los métodos, le dará el URI en el que se ejecutarán los métodos del controlador. A partir de esto se puede inferir que la asignación de solicitud a nivel de clase seguirá siendo la misma, mientras que cada método de controlador tendrá su propia asignación de solicitud. A nivel de método puede recibir también si es de tipo get, post, etc. Por ejemplo:

```
...

@RestController
@RequestMapping("/ws/clientes")
public class ClienteRestController {

  private static final Logger log =
LoggerFactory.getLogger(ClienteRestController.class);

  @Autowired
  private ClienteService clienteService;

  // Listar los clientes
  @RequestMapping(method = RequestMethod.GET)
  public List<ClienteDTO> findAll() {

    log.info("ClienteRestController - findAll: Mostramos todos los clientes");

    List<ClienteDTO> listaClientesDTO = clienteService.findAll();
    return listaClientesDTO;
  }

  // Visualizar la informacion de un cliente
  @RequestMapping(method = RequestMethod.GET, path = "/{idCliente}")
  public ClienteDTO findById(@PathVariable("idCliente") Long idCliente) {

    log.info("ClienteRestController - findById: Mostramos la informacion del
cliente:" + idCliente);

    // Obtenemos el cliente y lo pasamos al modelo
    ClienteDTO clienteDTO = new ClienteDTO();
    clienteDTO.setId(idCliente);
    clienteDTO = clienteService.findById(clienteDTO);
    return clienteDTO;
  }
}
```

En el primer método, denominado `findAll()` retornará la lista de objetos de tipo `ClienteDTO`, y la URI que ejecutará este método será:

http://localhost:8888/ws/clientes

mientras que el segundo método, denominado `findById(@PathVariable("idCliente") Long idCliente)` retornará un objeto de tipo `ClienteDTO`, y la URI que ejecutará dicho método será:

http://localhost:8888/ws/clientes/3

No usaremos este tipo de ejemplos sino que haremos uso de las siguientes anotaciones.

- `@GetMapping`, `@PostMapping`, `@PutMapping`, `@PathMapping`, `@DeleteMapping`: Estas anotaciones se agregan a los métodos. Cada anotación representa el tipo de método HTTP que se va a utilizar.
  - Get: para solicitar información de un recurso.
  - Post: para enviar información a fin de crear o de actualizar un recurso.
  - Put: para enviar información a fin de modificar un recurso.
  - Patch: actualiza una parte del recurso.
  - Delete: elimina un recurso específico.

  ¿Cuál es la diferencia entre Post , Put, Patch?

  Habitualmente la diferencia entre Post y Put radica en que Post lo usamos para **añadir** un recurso y Put lo utilizamos para **modificar** un recurso en particular.

  Patch también lo utilizamos para actualizar un recurso pero solo una **parcialidad** del mismo.
- `@Produces`: Indica el tipo MIME que enviará al cliente y se debe especificar por cada método. Por ejemplo: `@Produces({"application/json","application/xml"})`. Por defecto produce JSON.
- `@Consumes`: Indica el tipo MIME que puede aceptar. Por ejemplo, en el caso de insertar un nuevo Cliente, podemos aceptar un mensaje XML indicando lo siguiente en el método a procesar la petición: `@Consumes("application/xml")`.

Recordamos que también haremos uso de:

- `@PathParam`: Para especificar parámetros se utilizan los signos { }. Los parámetros se adjuntan al método utilizando la anotación `@PathParam`. Puede haber múltiples parámetros. Ej. `@Path("/personas/{tipo}/{id}")`
- `@QueryParam`: Permite procesar los parámetros de la URL. Para especificar parámetros HTTP se agregan después de signo ?, y para agregar varios se utilizar el signo &. Ejemplo:

  http://localhost:8888/ws/clientes?fechaInicio=01012012&fechaFin=31122012

7. Adaptando los DTO para Servicios Web

Los objetos (DTO) se parsean (o convierten) en JSON de forma automática. Para ello Spring usa el mapeador de objetos Jackson (*Jackson object mapper*) para transformar objetos DTO en objetos JSON.

El problema que tenemos con esta transformación es el problema que nos suele aparecer con los DTO: convertir objetos en JSON puede llevarnos a otro problema de recurrencia infinita, cuando el objeto contiene referencias cruzadas, como los métodos toString().

Hemos aprendido cómo evitar que los métodos `toString()` produzcan la excepción `StackOverflowException` usando, de la libreria `Lombok`, la anotación `@ToString.Exclude`, pero este arreglo no lo evitaría cuando lo convertimos a JSON. La solución se ofrece con nuevas anotaciones de la siguiente manera:

- `@JsonIgnore`: este atributo no será parseado. Establecerá a nulo el valor.
- `@JsonManagedReference`: indica que será mostrada la información del atributo en una sola dirección, pero no mostrará la información hacia atras. Esta anotación permitirá evitar que se forme un bucle infinito, indicando quien el el padre y el hijo.Será complementada con la siguiente:
- `@JsonBackReference`: Esta anotación es la anotación hija de la anterior, de forma que evita un bucle infinito. Es similar a `@JsonIgnore`.
- `@JsonIgnoreProperties({"property1", "property2"})` : Esta anotación es usada a nivel de clase y permite ignorar 1 o varias propiedades.

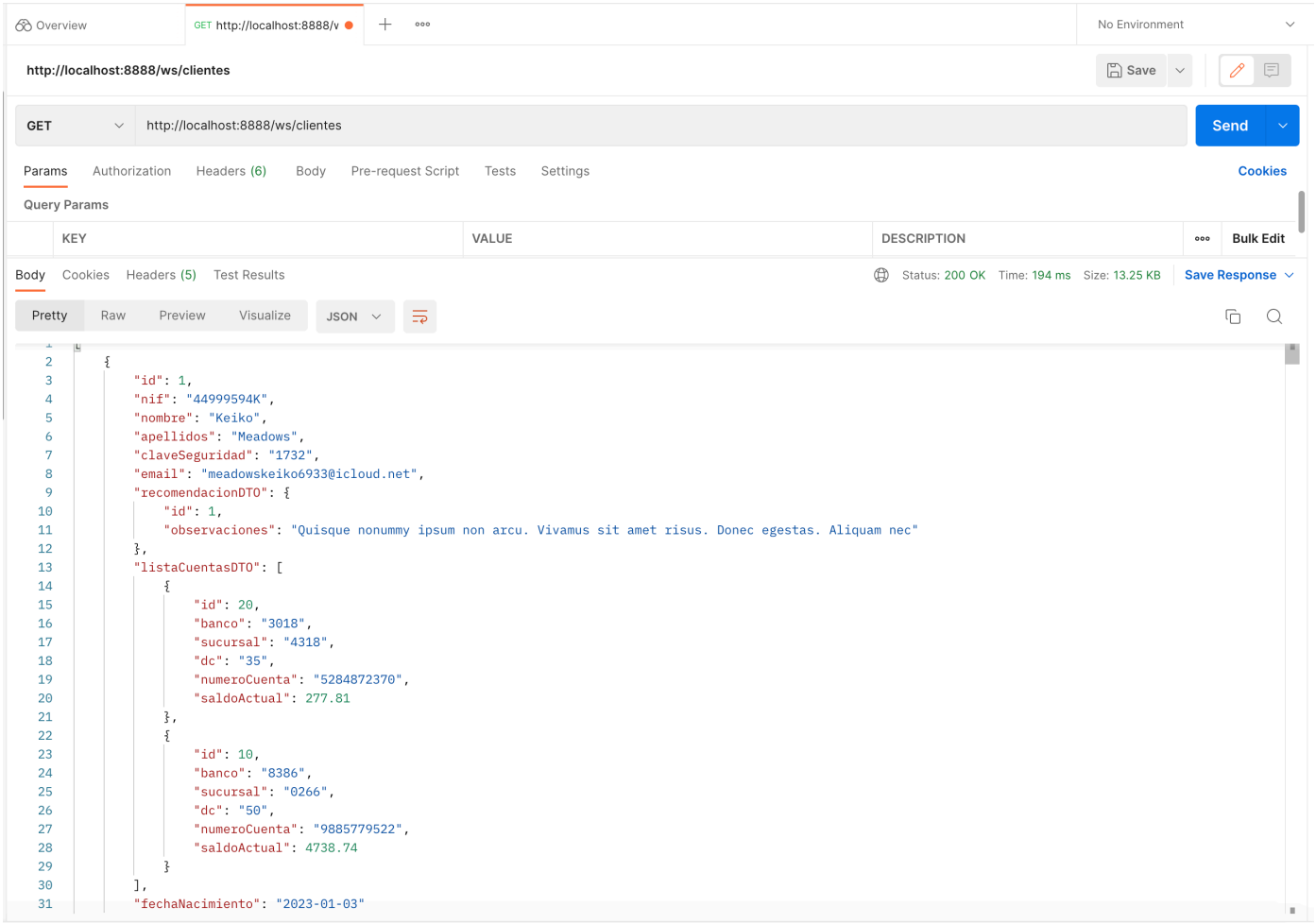
Un ejemplo de `@JsonManagedReference` y `@JsonBackReference` podría ser el siguiente:

```
...
@Data
public class ClienteDTO implements Serializable{

  private static final long serialVersionUID = 1L;
  private Long id;
  private String nif;
  private String nombre;
  private String apellidos;
  private String claveSeguridad;
  private String email;
  @ToString.Exclude
  @JsonManagedReference
  private RecomendacionDTO recomendacionDTO;
  @ToString.Exclude
  @JsonManagedReference
  private List<CuentaDTO> listaCuentasDTO;
  //@ToString.Exclude
  //private List<DireccionDTO> listaDireccionesDTO;

  @ToString.Exclude
  @JsonIgnore
  private List<ClienteDireccionDTO> listaClientesDireccionesDTO;
  ...
}
```

```
...
@Data
public class RecomendacionDTO implements Serializable{

  private static final long serialVersionUID = 1L;
  private Long id;
  private String observaciones;
  @ToString.Exclude
  @JsonBackReference
  private ClienteDTO clienteDTO;
  ...
}
```

```
...
@Data
public class CuentaDTO implements Serializable{

  private static final long serialVersionUID = 1L;
  private Long id;
  private String banco;
  private String sucursal;
  private String dc;
  private String numeroCuenta;
  private float saldoActual;
  @ToString.Exclude
  @JsonBackReference
  private ClienteDTO clienteDTO;
  ...
}
```

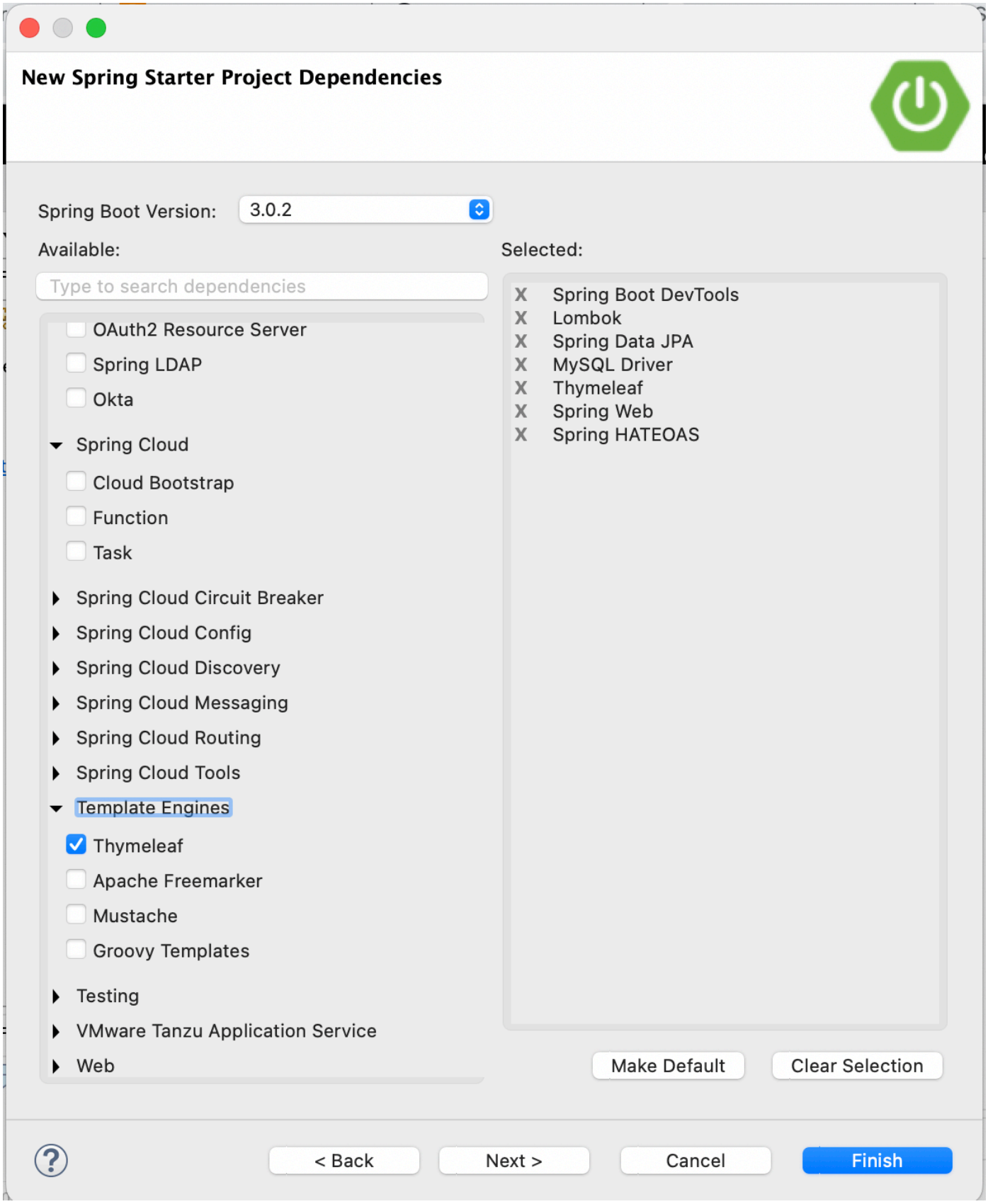
Si invocaramos una petición que obtuviera la lista de clientes podriamos comprobar que obtendriamos un JSON con la lista de clientes y mostraria la recomendacion y la lista de cuentas, pero no crearia una referencia circular de las mismas. El resultado sería este:



8. Integrando Servicios Web de tipo REST en nuestra aplicación Spring Boot

Vamos a seguir usando nuestro proyecto actual pero nuestro objetivo que perseguimos con la nueva modificación del programa es poder realizar operaciones web de tipo REST sobre los clientes. Para ello realizaremos lo siguiente:

1. Crearemos un nuevo proyecto, denominado demo7 el cual tendrá los siguientes starters:



2. Copiamos todo el código del proyecto `demo6` y nos lo traemos a `demo7`. La única clase que no nos traeremos será `Demo6Application.java`, ya que esta clase es la que arranca el proyecto.

3. Ahora vamos a solucionar las referencias cíclicas. Una referencia cíclica se produce cuando un objeto A apunta a otro objeto B y este objeto B apunta al objeto A. Al generar el fichero JSON el sistema no sabe cómo resolver esta referencia cíclica, puesto que estaría continuamente mapeando al JSON objeto A – objeto B - objeto A – objeto B ... y así sucesivamente.
4. En caso de estar produciendo XML, en cada clase DTO debemos poner, a nivel de clase la anotación `@XmlRootElement` y, para resolver las referencias cíclicas habladas con anterioridad, la anotación que permite que dichos atributos sean excluidos es `@XmlTransient`. Esto se cuenta pero no lo vamos a utilizar ya que vamos a producir JSON.
5. Como ya hemos indicado vamos a trabajar con JSON, por lo que vamos a añadir anotaciones en las clases `ClienteDTO.java` , `RecomendacionDTO.java` y `CuentaDTO.java`. El código de estas 3 clases lo hemos puesto cuando hemos hablado de las anotaciones, por lo que lo omitimos aquí.
6. Tras esto creamos el controlador `ClienteRestController.java` dentro del paquete `com.example.demo.web.webservice`.
7. Añadiremos las anotaciones `@RestController` y `@RequestMapping` al nuevo controlador. En el caso de la anotación `@RequestMapping` añadiremos la URL `/ws/clientes`, de forma que nuestro proyecto debe resolver las operaciones REST siempre con el prefijo `/ws`. Si no indicamos esto y hacemos que las operaciones sean resueltas en `/clientes` nos dará un error a la hora de ejecutarlo, puesto que en la URL `/clientes` tendremos a 2 controladores mapeando las mismas operaciones. Lo mejor para no tener problemas es añadir el prefijo indicado cuando son operaciones de tipo REST.
8. Vamos a implementar las siguientes operaciones (echaremos un vistazo al controlador `ClienteController.java` para orientarnos):
    1. Obtener todos los clientes: Método GET que devuelve toda la colección. Ya implementado en el método `findAll()` de `ClienteController.java`. Nos lo copiamos para despues adaptarlo.
    2. Obtener un cliente determinado: Método GET que devuelve un objeto determinado utilizando un parámetro Path. Ya implementado en el método `findById()` de `ClienteController.java`. Nos lo copiamos para despues adaptarlo.
    3. Registrar un nuevo cliente: Método POST que registra un nuevo cliente en la base de datos. Ya implementado con el método `add()` y `save()` de `ClienteController.java`. Nos los copiamos para despues adaptarlo.
    4. Modificar un cliente: Método PUT que modifica un cliente en la base de datos. Ya implementado con el método `update()` y `save()` de `ClienteController.java`. Nos los copiamos para despues adaptarlo.
    5. Eliminar un cliente: Método DELETE que elimina un cliente existente. Ya implementado en `delete()` de `ClienteController.java`. Nos los copiamos para despues adaptarlo.
9. Como veremos, algunas de las operaciones devuelven un error controlado (mediante un gestor de excepciones que se ha definido al final del controlador) cuando el cliente solicitado no existe. En esos casos, se devuelve además una respuesta definida en la clase `Response` para notificar el código de error y mensaje de negocio, a parte del código de estado correspondiente (ya hablado en el punto de las cabeceras de peticiones).
10. Concretando cada operación de REST un poco más:
    1. Cada método anotado define un endpoint que podrá ser invocado por otra aplicación

2. Las anotaciones `@GetMapping`, `@PostMapping`, `@PutMapping`, `@PatchMapping`, `@DeleteMapping` definen el método (GET, POST, PUT, PATCH, DELETE) y la URL de dicho endpoint.
3. Si el endpoint debe utilizar `Path Params` vendrán definidos en la URL y también en el método como `@PathVariable`
4. Si el endpoint debe utilizar `Query Params` vendrán definidos solamente en el método como `@RequestParam`
5. Si el endpoint debe utilizar `Body Params` vendrán definidos solamente en el método como `@RequestBody`
6. La respuesta será siempre un objeto `ResponseEntity` que contendrá la información de respuesta y un código de estado HTTP asociado.

8.1. Operaciones de lectura (GET)

Comenzaremos por el método de listar clientes, donde no recibe ni `Path Params` ni `Query Params` ni `Body Params`. Sería:

```
  // Listar los clientes
  @GetMapping("/clientes")
  public ResponseEntity<List<ClienteDTO>> findAll() {

    log.info("ClienteRestController - findAll: Mostramos todos los clientes");

    List<ClienteDTO> listaClientesDTO = clienteService.findAll();

    return new ResponseEntity<>(listaClientesDTO, HttpStatus.OK);
  }
```

Si nos fijamos el método es bastante sencillo.

Vamos a modificar el método `findAll()` con el objetivo de poder recibir un parámetro en la solicitud. En este caso vamos a recibir apellidos, ya sean de forma completa o parcial, para poder filtrar los clientes. Quedará de la siguiente forma:

```
  // Listar los clientes
  @GetMapping("/clientes")
  public ResponseEntity<List<ClienteDTO>> findAll(@RequestParam(value = "apellidos",
defaultValue = "") String apellidos) {

    List<ClienteDTO> listaClientesDTO = null;
    if(apellidos.isBlank()) {
      log.info("ClienteRestController - findAll: Mostramos todos los clientes");
      listaClientesDTO = clienteService.findAll();
    }else {
      log.info("ClienteRestController - findAll: Mostramos los clientes con apellidos "
+ apellidos);
```

```
      listaClientesDTO = clienteService.findByApellidos(apellidos);
    }

    return new ResponseEntity<>(listaClientesDTO, HttpStatus.OK);
  }
```

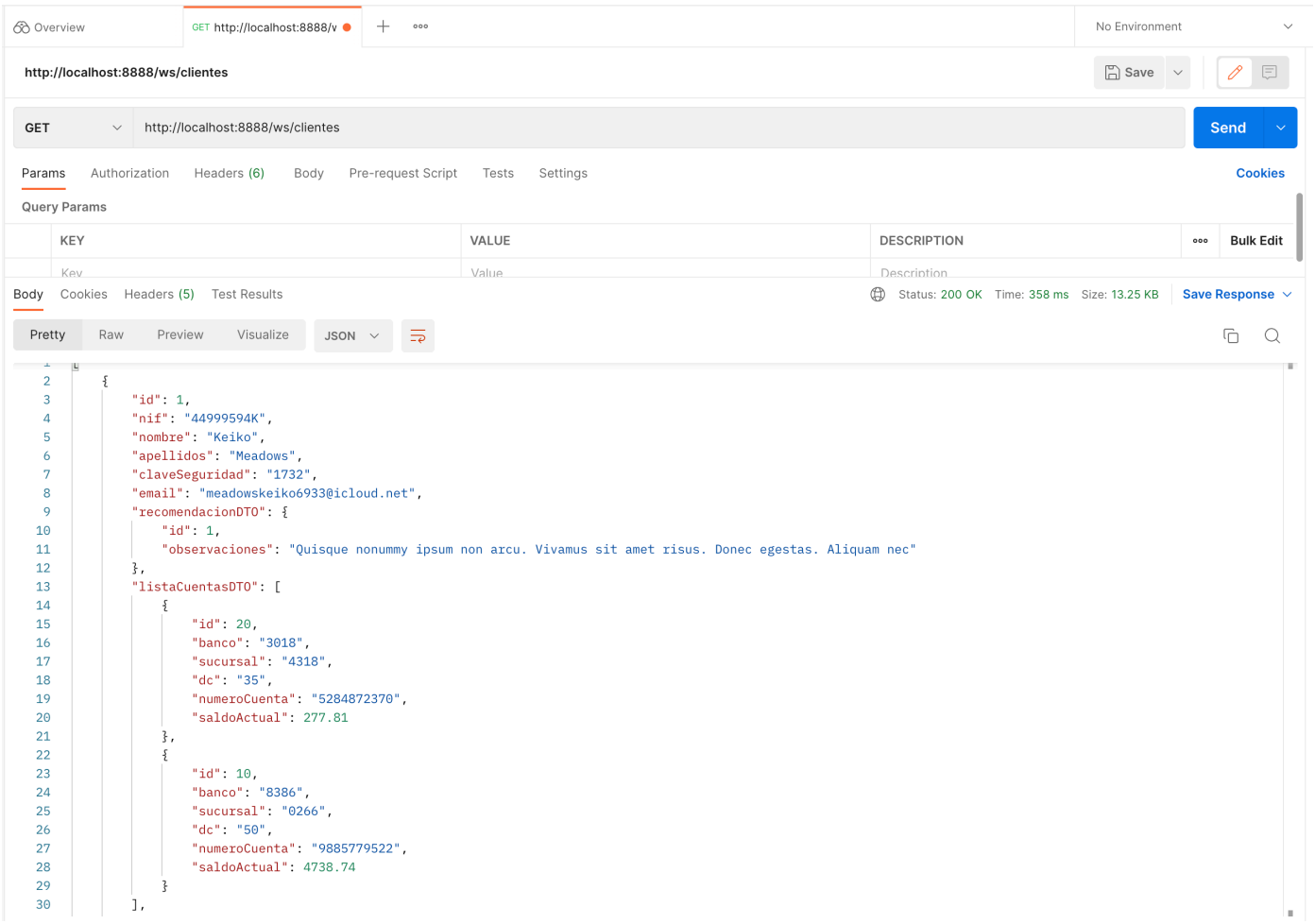
Fijaros que no tenemos implementado el método `clienteService.findByApellidos()` pero es muy sencilla su implementación, por lo que la omitimos.

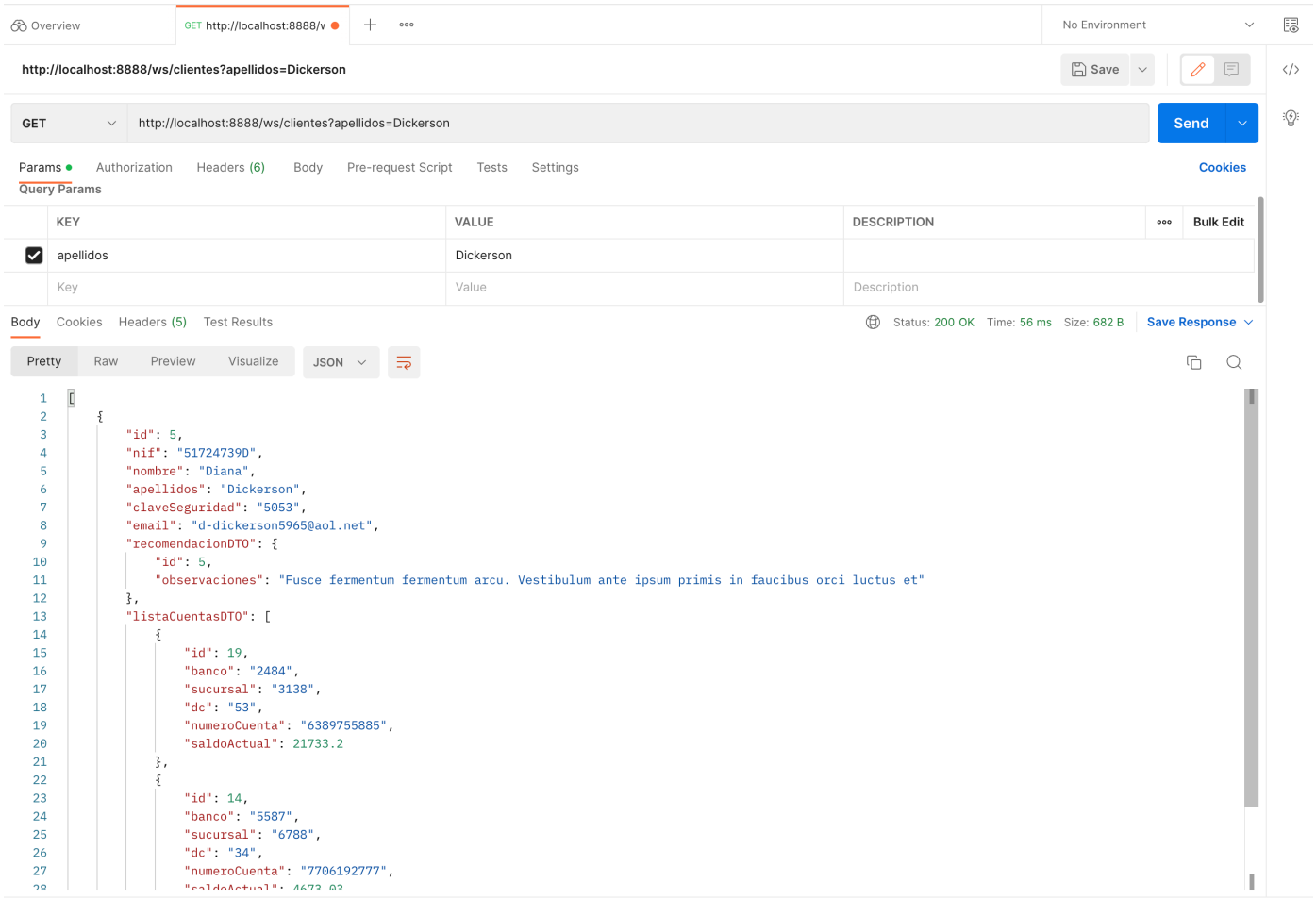
Recordamos que si en la anotación `@RequestMapping` a nivel de clase tiene el valor `"/ws/clientes"` la anotación `@GetMapping` no debe llevar `"/clientes"`.

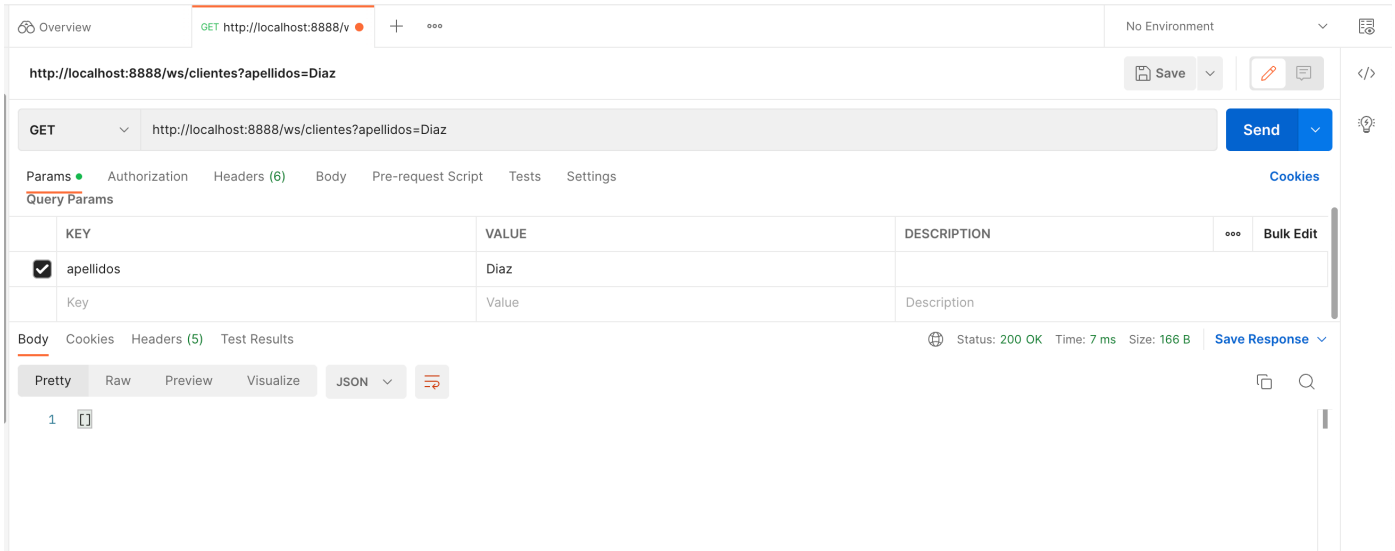
Ahora vamos a crear el método `findById()` de forma que obtendremos el ClienteDTO a traves de un id. En caso de no encontrarlo retornaremos un estado de no encontrado. Recordar que los resultados siempre son devueltos en el objeto `ResponseEntity`. Quedará de la siguiente forma:

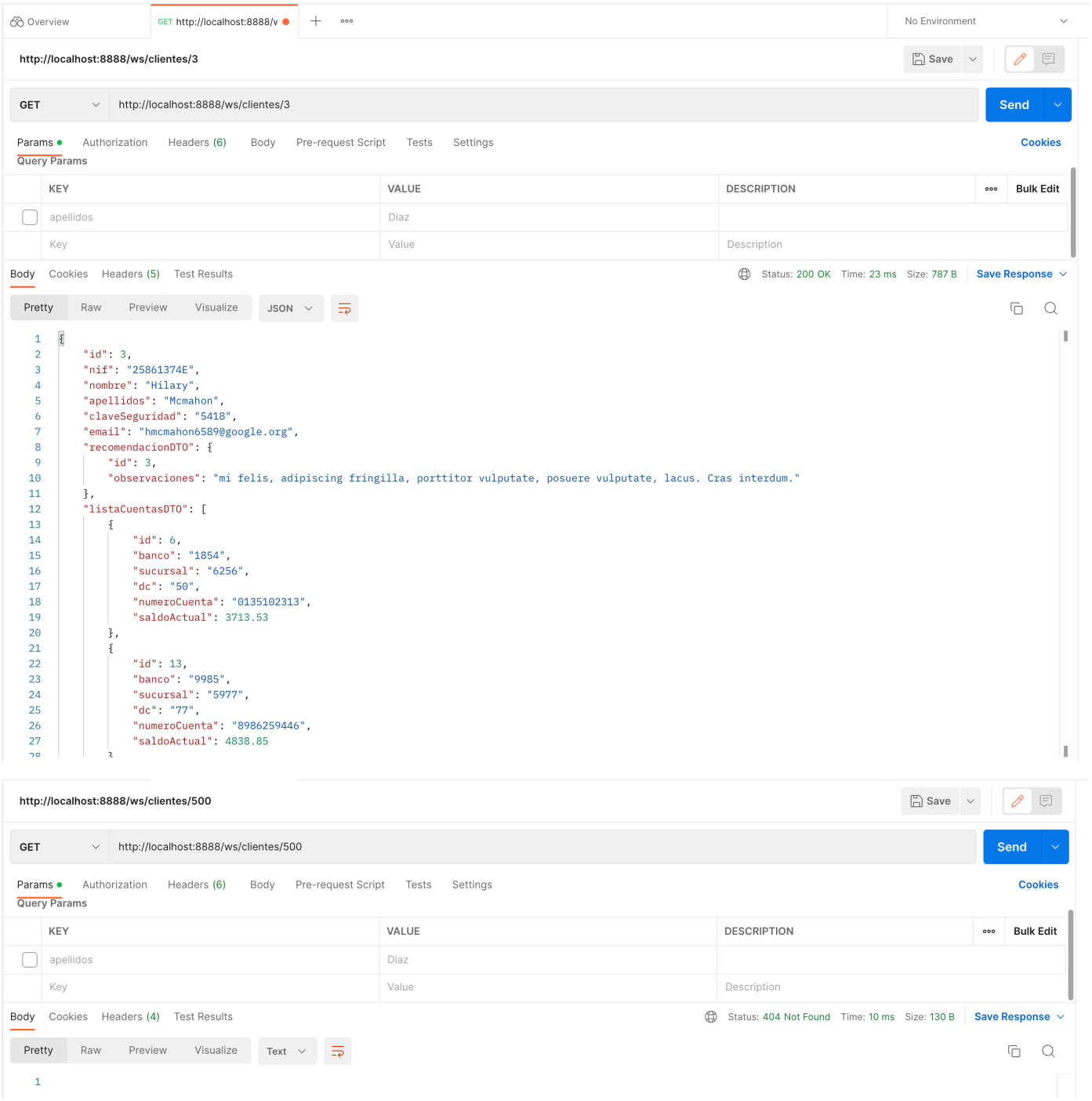
```
  // Localizamos un cliente por id
  @GetMapping("/clientes/{idCliente}")
  public ResponseEntity<ClienteDTO> findById(@PathVariable("idCliente") Long idCliente)
{

    log.info("ClienteRestController - findById: Localizamos el cliente con id:" +
idCliente);

    ClienteDTO clienteDTO = new ClienteDTO();
    clienteDTO.setId(idCliente);
    clienteDTO = clienteService.findById(clienteDTO);
    if(clienteDTO == null) {
      return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }else {
      return new ResponseEntity<>(clienteDTO, HttpStatus.OK);
    }
  }
```

Si ejecutamos en Postman las distintas operaciones obtenemos lo siguiente:


Overview
GET http://localhost:8888/v
No Environment
http://localhost:8888/ws/clientes
Save
GET
http://localhost:8888/ws/clientes
Send
Params
Authorization
Headers (6)
Body
Pre-request Script
Tests
Settings
Cookies
Query Params
KEY
VALUE
DESCRIPTION
Bulk Edit
Key
Value
Description
Body
Cookies
Headers (5)
Test Results
Status: 200 OK
Time: 358 ms
Size: 13.25 KB
Save Response
Pretty
Raw
Preview
Visualize
JSON
{
"id": 1,
"nif": "44999594K",
"nombre": "Keiko",
"apellidos": "Meadows",
"claveSeguridad": "1732",
"email": "meadowskeiko6933@icloud.net",
"recomendacionDTO": {
"id": 1,
"observaciones": "Quisque nonummy ipsum non arcu. Vivamus sit amet risus. Donec egestas. Aliquam nec"
},
"listaCuentasDTO": [
{
"id": 20,
"banco": "3018",
"sucursal": "4318",
"dc": "35",
"numeroCuenta": "5284872370",
"saldoActual": 277.81
},
{
"id": 10,
"banco": "8386",
"sucursal": "0266",
"dc": "50",
"numeroCuenta": "9885779522",
"saldoActual": 4738.74
}
],


Overview
GET http://localhost:8888/v
No Environment
http://localhost:8888/ws/clientes?apellidos=Dickerson
Save
GET
http://localhost:8888/ws/clientes?apellidos=Dickerson
Send
Params
Authorization
Headers (6)
Body
Pre-request Script
Tests
Settings
Cookies
Query Params
KEY
VALUE
DESCRIPTION
Bulk Edit
apellidos
Dickerson
Key
Value
Description
Body
Cookies
Headers (5)
Test Results
Status: 200 OK
Time: 56 ms
Size: 682 B
Save Response
Pretty
Raw
Preview
Visualize
JSON
[
{
"id": 5,
"nif": "51724739D",
"nombre": "Diana",
"apellidos": "Dickerson",
"claveSeguridad": "5053",
"email": "d-dickerson5965@aol.net",
"recomendacionDTO": {
"id": 5,
"observaciones": "Fusce fermentum fermentum arcu. Vestibulum ante ipsum primis in faucibus orci luctus et"
},
"listaCuentasDTO": [
{
"id": 19,
"banco": "2484",
"sucursal": "3138",
"dc": "53",
"numeroCuenta": "6389755885",
"saldoActual": 21733.2
},
{
"id": 14,
"banco": "5587",
"sucursal": "6788",
"dc": "34",
"numeroCuenta": "7706192777",


Overview
GET http://localhost:8888/v
No Environment
http://localhost:8888/ws/clientes?apellidos=Diaz
Save
GET
http://localhost:8888/ws/clientes?apellidos=Diaz
Send
Params
Authorization
Headers (6)
Body
Pre-request Script
Tests
Settings
Cookies
Query Params
KEY
VALUE
DESCRIPTION
Bulk Edit
apellidos
Diaz
Key
Value
Description
Body
Cookies
Headers (5)
Test Results
Status: 200 OK
Time: 7 ms
Size: 166 B
Save Response
Pretty
Raw
Preview
Visualize
JSON
[]



8.2. Operaciones de inserción (POST)

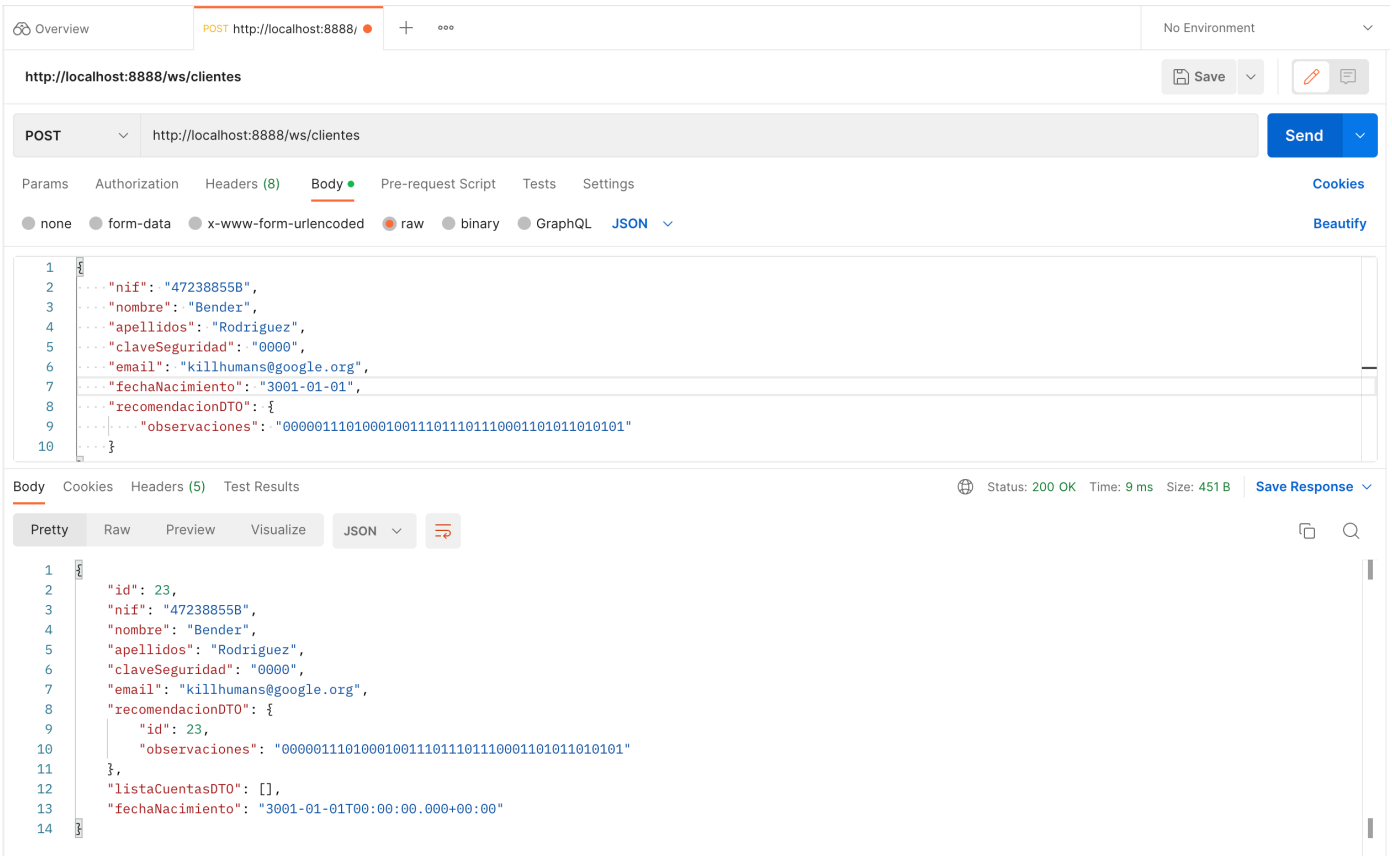
Se trata de la primera operación que necesita almacenar un nuevo cliente en la base de datos. El controlador recibe un nuevo objeto de tipo DTO que es enviado mediante una aplicación (en este caso usamos PostMan para enviar el objeto).

```java
// Alta de clientes
@PostMapping("/clientes")
public ResponseEntity<ClienteDTO> add(@RequestBody ClienteDTO clienteDTO) {

  log.info("ClienteRestController - add: Anyadimos un nuevo cliente");

  clienteDTO = clienteService.save(clienteDTO);
  if(clienteDTO == null) {
    return new ResponseEntity<>(HttpStatus.BAD_REQUEST);
  }else {
    return new ResponseEntity<>(clienteDTO, HttpStatus.OK);
  }
}
```

La solicitud es mapeada mediante `@PostMapping`, donde el objeto viene como un JSON en el cuerpo de la petición, y nosotros usamos `@RequestBody` para recogerlo. Suponemos que el objeto viene de forma correcta y que no tiene ningún problema.

Con esto enviamos el objeto a ser almacenado al servicio. Hemos modificado esta capa también para que, una vez almacenado el objeto lo devuelva, así dispondremos del objeto almacenado con su id (esta modificación no afecta a nada de la capa de visualización usada con Thymeleaf).

Las peticiones en PostMan serían:



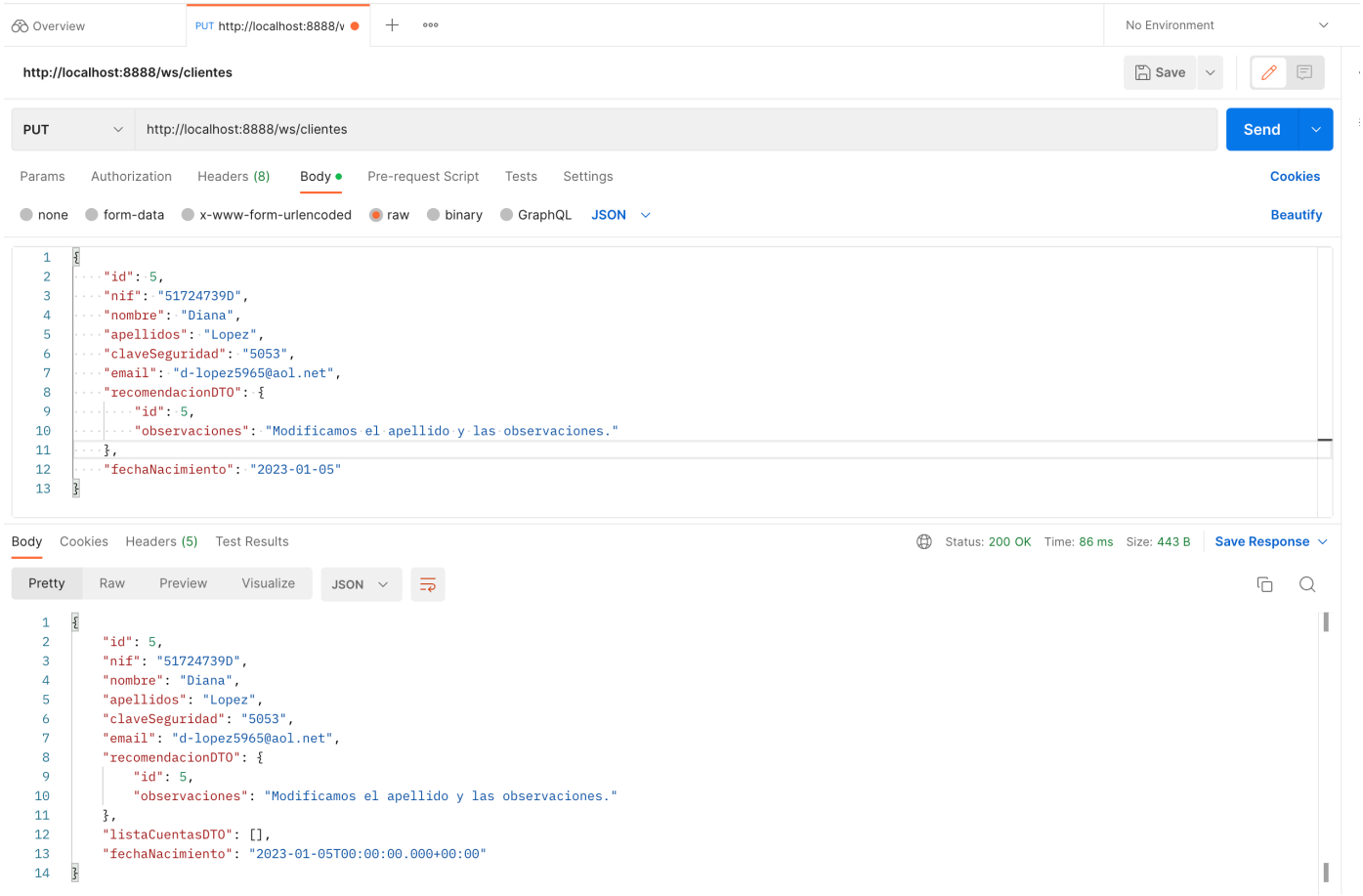
Llegados a este punto nos damos cuenta que el JSON envía nombres como `recomendacionDTO` o `listaCuentasDTO`. Estos nombres pertenecen a toda la parte de desarrollo interna y ahora un desarrollador externo, al querer usar nuestras APIS (o nuestros endpoints) quedan expuestas con estos nombres de variables. Quizas los nombres correctos deberian haber sido "`recomendacion`" y "`cuentas`".

8.3. Operaciones de Actualización (UPDATE)

Para actualizar un objeto de la base de datos, necesitamos recibir el objeto actualizado (y completo) en la solicitud. No podemos guardarlo inmediatamente, porque podría no existir. Por ello debemos comprobar la existencia, y si es positivo, guardar el objeto recibido, que actualizará la versión anterior en la base de datos. Vamos a ver como sería la operación:

```
  // Actualizacion de clientes
  @PutMapping("/clientes")
  public ResponseEntity<ClienteDTO> update(@RequestBody ClienteDTO clienteDTO) {

    log.info("ClienteRestController - update: Modificamos el cliente: " +
clienteDTO.getId());

    // Obtenemos el cliente para verificar que existe
    ClienteDTO clienteExDTO = new ClienteDTO();
    clienteExDTO.setId(clienteDTO.getId());
    clienteExDTO = clienteService.findById(clienteExDTO);

    if(clienteExDTO == null) {
      return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }else {
      clienteDTO = clienteService.save(clienteDTO);
      return new ResponseEntity<>(clienteDTO, HttpStatus.OK);
    }
  }
```

Las peticiones en PostMan serían:



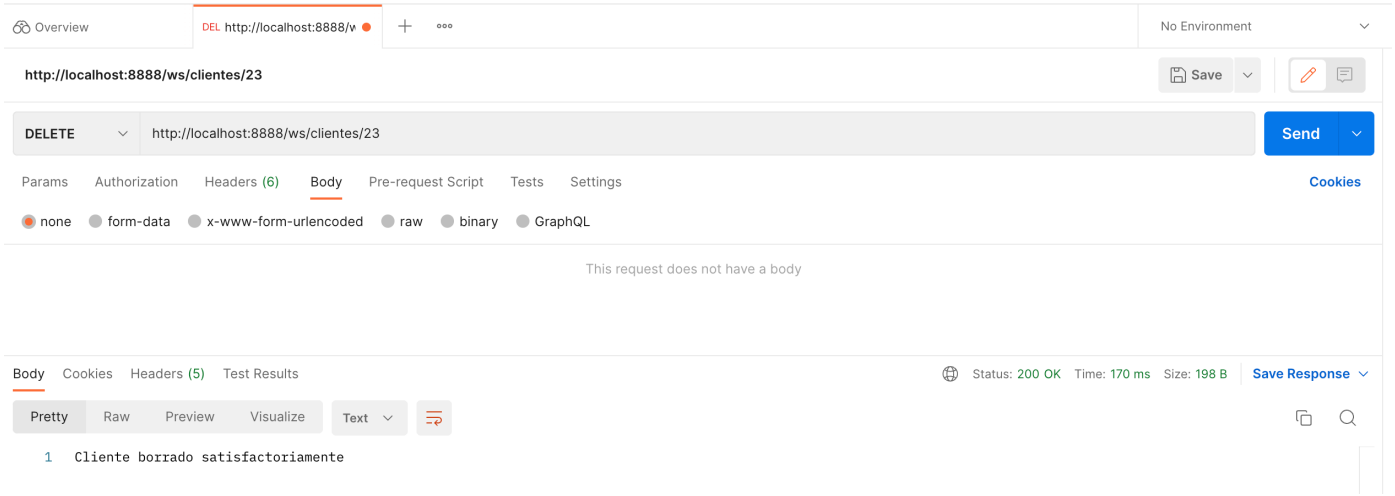
Podriamos haber utilizado `@PatchMapping` perfectamente, de hecho es más recomendable utilizarlo, puesto que solo modificaría los campos que son necesarios.

8.4. Operaciones de borrado (DELETE)

Borrar es una operación muy sencilla, ya que solo necesitamos el identificador del objeto que queremos borrar, que vendrá en la URI como una variable de ruta. La operación quedará de la siguiente forma:

```
// Borrar un cliente
@DeleteMapping("/clientes/{idCliente}")
public ResponseEntity<String> delete(@PathVariable("idCliente") Long idCliente) {

  log.info("ClienteRestController - delete: Borramos el cliente:" + idCliente);

  // Creamos un cliente y le asignamos el id. Este cliente es el que se va a borrar
  ClienteDTO clienteDTO = new ClienteDTO();
  clienteDTO.setId(idCliente);
  clienteService.delete(clienteDTO);

  return new ResponseEntity<>("Cliente borrado satisfactoriamente", HttpStatus.OK);
}
```

Hemos programado un borrado sencillo. Si lo invocamos a través de Postman obtenemos lo siguiente:


Overview
DEL http://localhost:8888/v
No Environment
http://localhost:8888/ws/clientes/23
Save
DELETE
http://localhost:8888/ws/clientes/23
Send
Params
Authorization
Headers (6)
Body
Pre-request Script
Tests
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
This request does not have a body
Body
Cookies
Headers (5)
Test Results
Status: 200 OK
Time: 170 ms
Size: 198 B
Save Response
Pretty
Raw
Preview
Visualize
Text
1 Cliente borrado satisfactoriamente