Java Spring Boot

Unidad 2. Servicios REST con Spring

Índice

Unidad 2: Servicios REST con Spring

- Introducción
- Objetivos
- Mapa conceptual
- El framework Spring
- Servicios Rest con Spring
- Creación de un servicio Rest
- Probando el servicio
- Objetos complejos
- Variables en URL
- Otros métodos HTTP
- Aplicación Postman para testing de servicio

Ejercicio práctico y solución

Autoevaluación

Introducción

Uno de los principales frameworks o conjunto de utilidades para la creación de aplicaciones Java es Spring. Spring se compone de varios módulos, especializados en la creación de un determinado componente o capa de aplicación. Y, como no podía ser de otra manera, algunos de esos módulos nos van simplificar la construcción de servicios REST.

La creación de servicios REST con Spring es una tarea sumamente sencilla, pues permite al programador centrase en la lógicade la aplicación y dejarle al framework los detalles de interacción con los clientes.

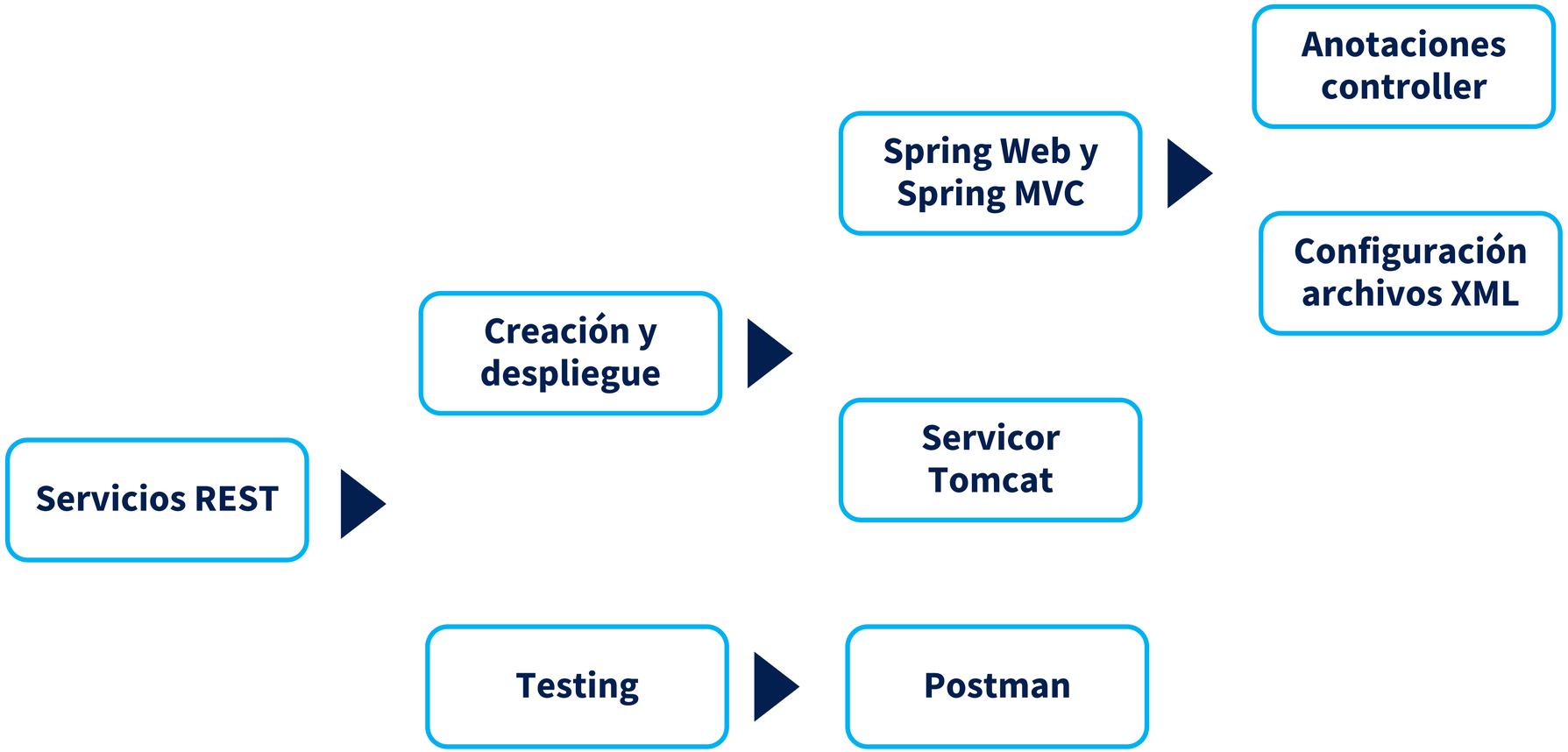
Objetivos

Revisar los principales módulos y características del framework Spring.

Aplicar los componentes Spring en la creación de servicios REST.

Testear servicios REST.

Mapa Conceptual



El Framework Spring

Fundamentos

Spring es uno de los frameworks o entornos de trabajo para la creación de aplicaciones Java más populares. Se encuentra dividido en módulos, facilitando su uso dentro de las aplicaciones.

Son muchas las características y beneficios que aporta Spring a la hora de realizar los desarrollos, pero fundamentalmente, se basa en una idea principal y es la de **menos código a costa de más configuración**. Es decir, la idea es que los programadores se centren en desarrollar la lógica de la aplicación, dejando al framework la realización de tareas rutinarias y pesadas, aunque para ello, habrá que indicarle de forma declarativa todo aquello que debe realizar.

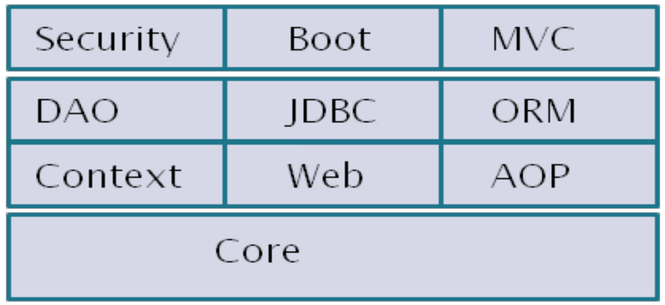
Además de la reducción de código, otros conceptos importantes como la **inyección de dependencias** o el **desacoplamiento entre capas de la aplicación**, son piezas esenciales en la construcción de aplicaciones Spring.

El Framework Spring

Módulos

Como hemos indicado, Spring es un framework organizado en diferentes módulos. A la hora de desarrollar una aplicación, podemos cargar solo aquellos módulos que necesitemos para el desarrollo.

En la siguiente imagen, tenemos la pila de módulos más importantes que forman Spring.



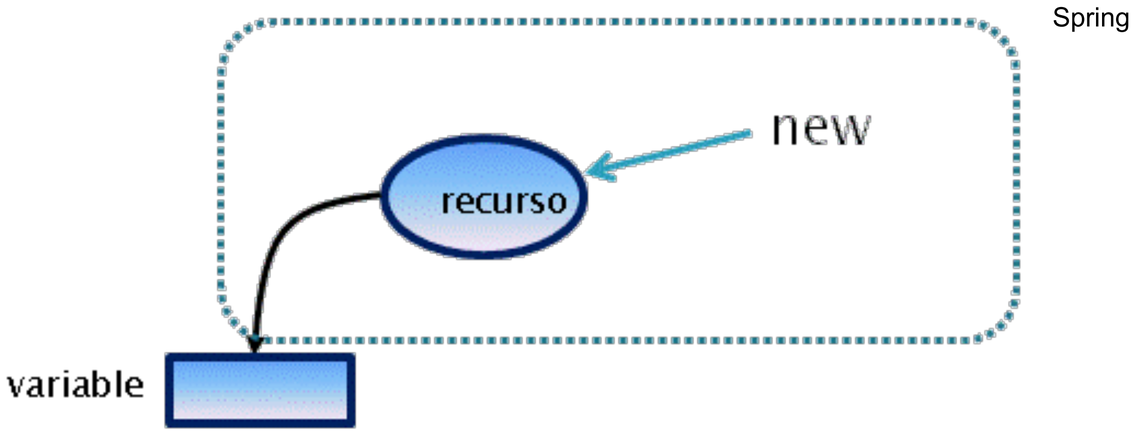
Según se desprende de la imagen anterior, el **módulo "Core" representa la base del framework**. Este módulo nos proporciona las características básicas, tales como la **inyección de dependencias**. Independientemente del tipo de aplicación que creemos, el módulo Core siempre estará presente.

El Framework Spring

Inyección de dependencias

La **inyección de dependencias** es un concepto clave en Spring. La idea es que, cuando un objeto necesite otro objeto, conocido como recurso, para realizar su trabajo, no sea el programador el que se tenga que encargar desde código de crear y configurara el objeto, sino que se delegue esta tarea en el framework

Spring será el encargado de, en función de los metadatos indicados, **crear, configurar e inyectar el recurso** en la variable correspondiente.



El Framework Spring

Configuración

Como hemos indicado, gran parte del trabajo de creación de una aplicación Spring está en la configuración de la misma, a través de la cual se le inda al framework lo que debe hacer.

La configuración de aplicaciones Spring se realiza mediante la combinación de los siguientes elementos:

• **Archivos de configuración XML**. Cada bloque o capa de la aplicación lleva su propio archivo XML, en ellos se le suele indicar a Spring los objetos que tiene que instanciar y como configurarlos. En aplicaciones grandes, el uso de estos ficheros puede llegar a ser pesado y tedioso. Como ya veremos más adelante, **Spring Boot elimina la utilización de ficheros XML** de configuración, simplificando aún más el desarrollo de las mismas.

• **Anotaciones**. El uso de anotaciones permite indicar a Spring donde tiene que inyectar los recursos y el **comportamiento** que deben tener ciertas clases y métodos. En el caso de servicios REST, utilizaremos bastantes anotaciones en la capa del adaptador Web.

El Framework Spring

Principales módulos

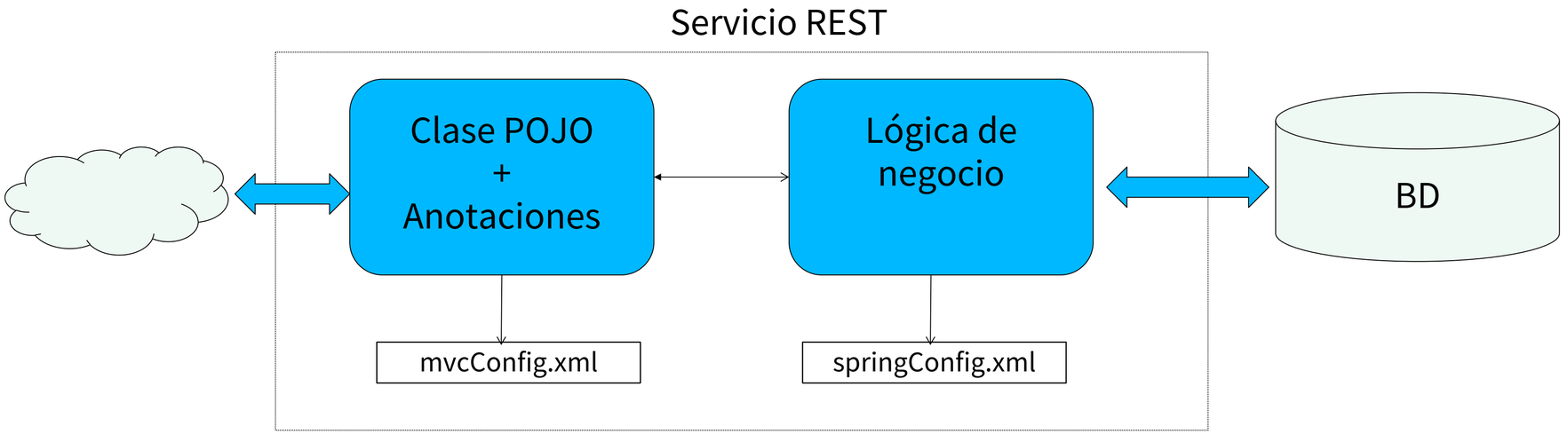
A continuación, comentamos algunos de los módulos principales que componen el framework:

- **Context. Módulo de apoyo a core en las tareas de inyección de dependencias**
- **Web. Componentes para utilizar Spring en aplicaciones Web, incluidos servicios REST**
- **JDBC, DAO y ORM. Acceso a datos**
- **AOP. Programación orientada a aspectos**
- **Security. Aplicación de políticas de seguridad en el acceso a las aplicaciones**
- **MVC. Aplicaciones Modelo Vista Controlador y servicios REST**
- **Boot. Configuración y despliegue de servicios**

Servicios REST con Spring

Fundamentos

Como sabemos, un servicio REST no es más que una aplicación Web que, en lugar de servir páginas a una aplicación, lo que hace es ofrecer recursos (datos y operaciones) a otros programas:



La capa de lógica de negocio será implementada con los **módulos Core, DAO/JPA**, y en general con todo aquello necesario para realizar su función. Pero lo que realmente caracteriza al servicio es el **módulo adaptador**, que en Spring será implementado mediante una **clase POJO formada por una serie de anotaciones** para indicar al framework lo que debe hacer. Tanto un módulo como el otro, dispondrán de sus correspondientes **archivos de configuración.**

Servicios REST con Spring

Anotaciones principales

La creación del servicio REST se va a centrar en lo que sería la **clase POJO que forma el adaptador Web o controlador** entre el cliente final y nuestra lógica de negocio. Cada recurso que queramos exponer, será implementado mediante un método. Tanto la clase como los métodos, tendrán que ser declarados con una serie de anotaciones. A continuación indicamos la lista de anotaciones más importantes, que serán analizadas con detalle más adelante:

- **@RestController**. Se utiliza en la definición de la clase para indicarle a Spring que se trata de un controlador REST
- **@GetMapping, @PostMapping, @PutMapping, @DeleteMapping**. Permiten asociar un método de la clase a un determinado método de petición HTTP (GET, POST, PUT y DELETE).
- **@PathVariable**. Mapea una varable de la URL con un parámetro del método de respuesta.
- **@RequestParam**. Mapea un parámetro de la petición con un parámetro del método de respuesta.
- **@RequestBody**. Mapea el dato recibido en el cuerpo de una petición con un objeto Java.

Creación de un servicio REST básico

Fundamentos

A continuación, vamos a explicar el proceso de creación de un servicio REST con Spring, utilizando el IDE Eclipse con **Maven para la gestión de dependencias**.

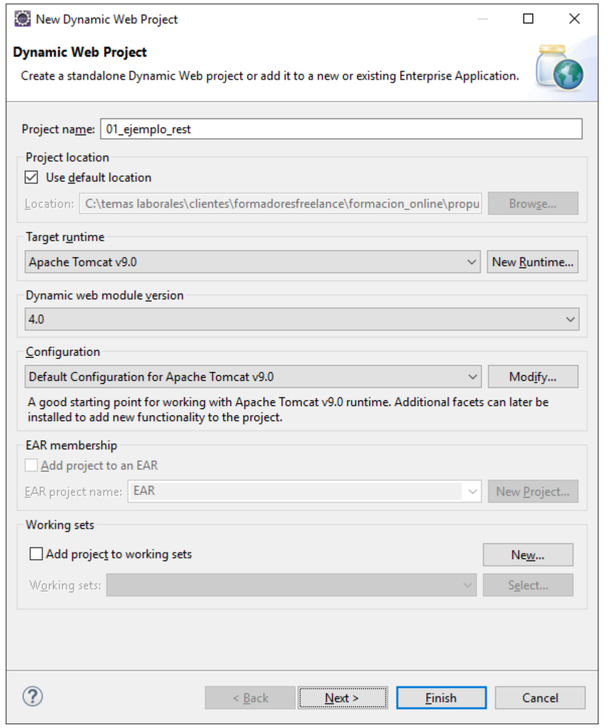
Para ello, vamos a crear un servicio sencillo que atienda a una petición de HTTP de tipo GET en una determinada URL, de modo que al recibir dicha petición devuelva al cliente una cadena de texto con un mensaje de saludo.

Creación de un servicio REST básico

Creación del proyecto

Una vez iniciado Spring, crearemos un proyecto Web dinámico utilizando la opción de menú *File->New->Dynamic Web Project*. Si no apareciera esta opción, es que no estamos en la perspectiva Web o Java EE, en cuyo caso tendríamos que seleccionar *File-New-Other* y dentro de la categoría Web elegir la opción Dynamic Web Project.

En el cuadro de dialogo que aparece a continuación, le daremos un **nombre al proyecto** y dejaremos el resto de opciones por defecto, verificando que tenemos seleccionado el servidor Tomcat que previamente hemos asociado a Eclipse:

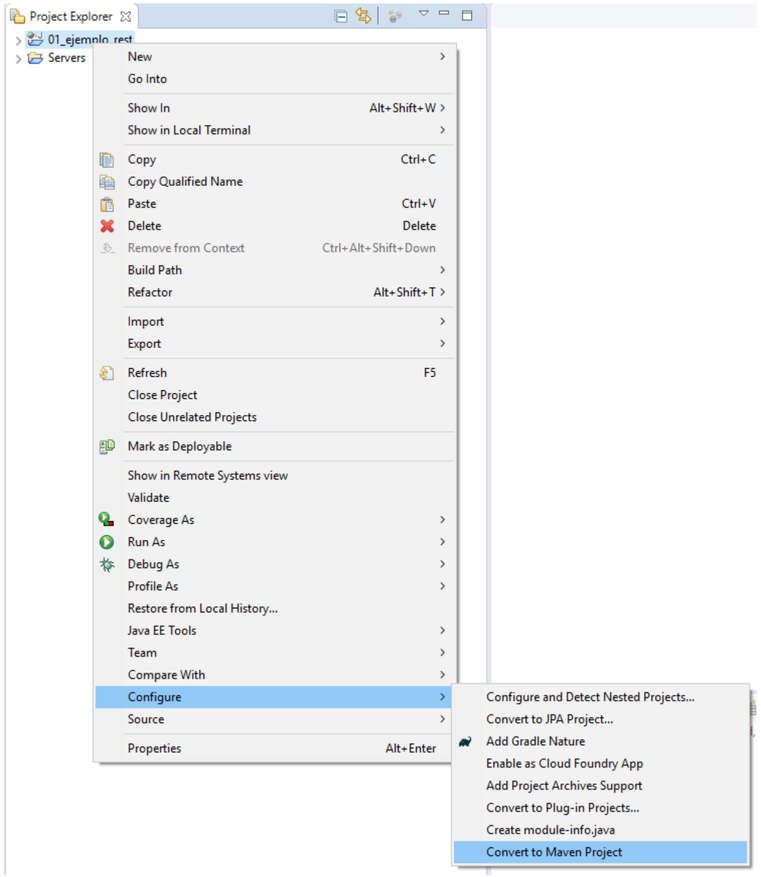


Creación de un servicio REST básico

Conversión a proyecto Maven

Una vez finalizada la creación del proyecto, lo primero que tendremos que hacer será **agregar las librerías necesarias para trabajar con Spring REST**. Como vamos a utilizar Maven para la gestión de dependencias, lo primero que haremos será convertir el proyecto en un proyecto **Maven**. Esto lo hacemos situándonos encima del proyecto en el explorador de proyectos ý, pulsando el botón derecho del ratón, elegimos la opción *Configure->Convert to Maven Project*.

Después de esto, aparecerá un cuadro de diálogo informando de las opciones de Maven y la estructura del **pom.xml**. Lo aceptamos directamente y veremos como en nuestro proyecto aparece el archivo pom.xml



Creación de un servicio REST básico

Dependencias

Ahora tenemos que agregar las dependencias en el archivo pom.xml. Estas dependencias podemos ir a buscarlas al repositorio de Maven o las puedes agregar directamente según lo que se indica en el siguiente bloque XML, el cual **será incluido dentro del pom** al final del mismo, entre las etiquetas de cierre </build> y </project>

Como vemos, las cuatro primeras dependencias corresponden a los módulos de Spring que vamos a utilizar, *core* y *context* que son los básicos y *web* y *webmvc* que contienen los elementos para la construcción del servicio Web.

Las dos últimas dependencias, que en este primer ejemplo no las vamos a utilizar, son dos librerías en las que se apoya Spring para el **mapeo de objetos Java a XML/JSON**

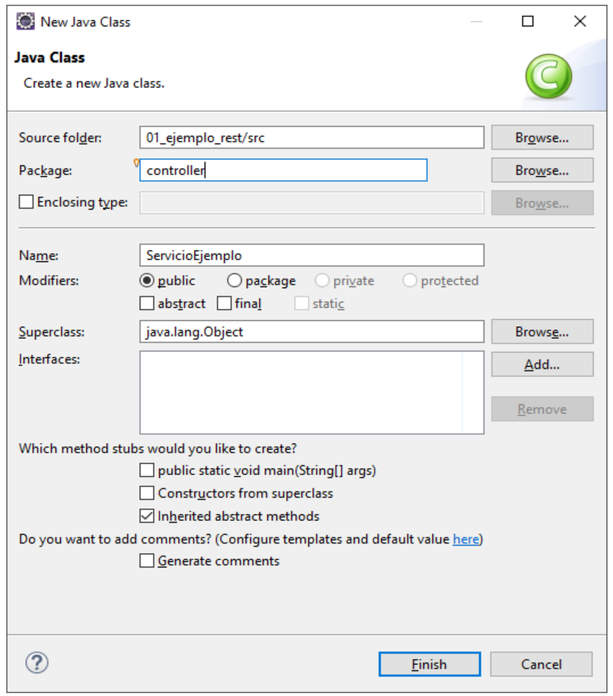
```
 <dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.1.5.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>5.1.5.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
    <version>5.1.5.RELEASE</version>
</dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.1.5.RELEASE</version>
</dependency>
<!-- mapeo objeto java a JSON -->
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-core</artifactId>
    <version>2.9.8</version>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.9.8</version>
</dependency>
  </dependencies>
```

Creación de un servicio REST básico

Clase controladora

Una vez agregadas las dependencias, es el momento de programar nuestro servicio, el cual consistirá simplemente en la clase controladora o adaptador Web. Para ello, nos situamos sobre el proyecto, pulsamos el botón derecho del ratón y elegimos New ->Class.

En el cuadro de diálogo que aparece a continuación, le daremos un nombre a la clase y al paquete en el que la vamos a incluir.



Creación de un servicio REST básico

Clase controladora

A continuación, implementaremos dentro de la clase el método que representa el recurso que vamos a exponer. Se trata de un método, cuyo nombre puede ser cualquiera, que no recibirá ningún parámetro y cuyo tipo de devolución será un String, dado que el mensaje de saludo devuelto por el servicio en la petición GET será una cadena de caracteres.

El código de la clase será el que se muestra en el siguiente listado:

```
package controller;
public class ServicioEjemplo {
  public String saludar() {
    return "bienvenido a mi servicio";
  }
}
```

Como vemos, se trata de código Java muy simple, en el que no hemos tenido que programar ninguna instrucción relativa a la comunicación HTTP con el exterior, de todo ello se encarga Spring. Pero para que Spring sepa como tratar esta clase, faltan incluir las anotaciones REST.

Creación de un servicio REST básico

Anotaciones

En el siguiente listado mostramos la clase completa, incluyendo las diferentes anotaciones Spring REST que le indican al Framework el comportamiento de nuestro servicio:

```
@RestController
public class ServicioEjemplo {
  @GetMapping(value="saludo",produces=MediaType.TEXT_PLAIN_VALUE
)
  public String saludar() {
    return "bienvenido a mi servicio";
  }
}
```

A través de la anotación **@RestController**, indicamos a Spring que se trata de una clase controladora REST, con lo que el framework será el encargado de instanciarla y utilizarla para atender a las peticiones clientes.

Mediante **@GetMapping**, indicamos a Spring que el método *saludar()* debe ser ejecutado al recibir una petición GET, cuya url sea “saludo”, tal y como se indica en el atributo *value*. Mediante el atributo *produces*, le indicamos el tipo de la respuesta a enviar al cliente.

Creación de un servicio REST básico

Configuración

A nivel de código, ya no tenemos que hacer nada más en la clase controladora. Lo que nos falta ahora es la tarea de **configuración de la aplicación Web**, que implica la creación de dos archivos. El primero de ellos es el **web.xml**, que es el archivo de **configuración de aplicaciones Web Java EE**, y el que habrá que registrar el servlet **DispatcherServlet** que se encarga de interceptar e interpretar todas las peticiones HTTP. Así deberá quedar el archivo web.xml, que debe ser creado dentro del directorio WebContent\WEB-INF del proyecto:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd" version="4.0">
  <display-name>02_pedidos_spring</display-name>
  <servlet>
   <servlet-name>Dispatcher</servlet-name>
   <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
   <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/mvcConfig.xml</param-value>
   </init-param>
  </servlet>
  <servlet-mapping>
   <servlet-name>Dispatcher</servlet-name>
   <url-pattern>/</url-pattern>
  </servlet-mapping>
</web-app>
```

Creación de un servicio REST básico

Configuración

En el archivo web.xml vemos que se hace referencia a otro archivo mvcConfig.xml, que es el segundo archivo de configuración de la aplicación y el que debe contener la **configuración de Spring** Web. Además del elemento *annotation-driven*, que permite el uso de anotaciones en esta capa, debemos incluir ***component-scan***, que le indica a Spring los paquetes donde se encuentran **las clases que debe instanciar**, en nuestro caso, el controlador:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xmlns:mvc="http://www.springframework.org/schema/mvc"
   xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-
beans.xsd
     http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-
4.3.xsd
     http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc-4.3.xsd">
   <mvc:annotation-driven/>
   <context:component-scan base-package="controller"/>

</beans>
```

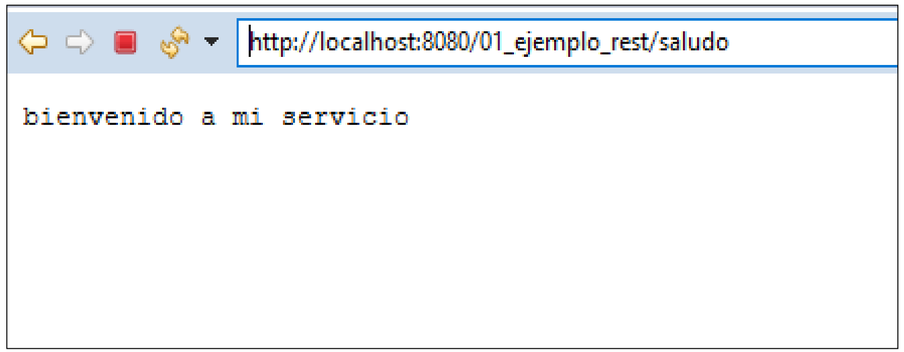
Probando el servicio

Después de crear los archivo de configuración. Ya tenemos listo el servicio para ser desplegado y utilizado. Al ejecutarlo desde el IDE, se inicia el servidor Tomcat y la aplicación es desplegada en él.

Como solamente hemos expuesto un recurso, accesible a través de una petición HTTP de tipo GET, podemos **probarlo desde el propio navegador** introduciendo la url:

http://localhost:8080/01_ejemplo_rest/saludo

El resultado será el que se muestra a continuación:



Trabajando con tipos complejos

Mapeo objeto-JSON

En el ejemplo anterior el recurso simplemente devuelve una cadena de texto, pero, ¿Qué sucede cuando tiene que devolver un objeto complejo, tipo JavaBean, o incluso una colección o array de estos objetos?. La solución a esta cuestión la proporciona directamente Spring, encargándose de **mapear automáticamente los objetos JavaBean a JSON y viceversa** con el apoyo de la librería jackson.

Por ejemplo, supongamos que en lugar de devolver un simple mensaje de texto queremos responder con un JSON que contenga los datos de una persona. Para ello, vamos a crear un JavaBean Persona en nuestro proyecto, que encapsule el nombre, email y edad de una persona. Crearemos esta clase en el paquete beans:

```
public class Persona {
   private String nombre;
   private String email;
   private int edad;
   public Persona(String nombre, String
email, int edad) {
      super();
      this.nombre = nombre;
      this.email = email;
      this.edad = edad;
   }
   public Persona() {}
   public String getNombre() {
      return nombre;
   }
   public void setNombre(String nombre) {
      this.nombre = nombre;
   }
   public String getEmail() {
      return email;
   }
   public void setEmail(String email) {
      this.email = email;
   }
   public int getEdad() {
      return edad;
   }
   public void setEdad(int edad) {
      this.edad = edad;
   }
}
```

Trabajando con tipos complejos

Mapeo objeto-JSON

Ahora, vamos a modificar el método de la clase controladora para que en lugar de devolver una cadena de caracteres, devuelva un objeto Persona con los datos de una persona cualquiera. Como al cliente no le puede devolver un objeto Java, le diremos a Spring a través del **parámetro *produces* de la anotación @GetMapping**, que lo convierta a un objeto JSON.

Así deberá quedar ahora el controlador:

```
@RestController
public class ServicioEjemplo {
   @GetMapping(value="saludo",produces=MediaType.APPLICATION_JSON_VALUE)
   public Persona saludar() {
      return new Persona("Ejemplo","test@gmail.com",25);
   }
}
```

Si lanzamos de nuevo la petición al navegador, obtendremos como respuesta un objeto JSON con los datos de la Persona. Como vemos, no hemos tenido que programar nada para realizar esta conversión, **Spring se ha encargado automáticamente de ello**. Si en lugar de un uno objeto el método devuelve una lista o array de objetos, Spring se encargaría de transformarlo en un array de objetos JSON.

Variables en URL

En servicios REST es habitual pasar datos como parte de la URL. Estos datos son conocidos como **variables URL**, y se enviarían separándolos con el carácter "/" del resto de la dirección y de otras variables:

`url/variable1/variable2`

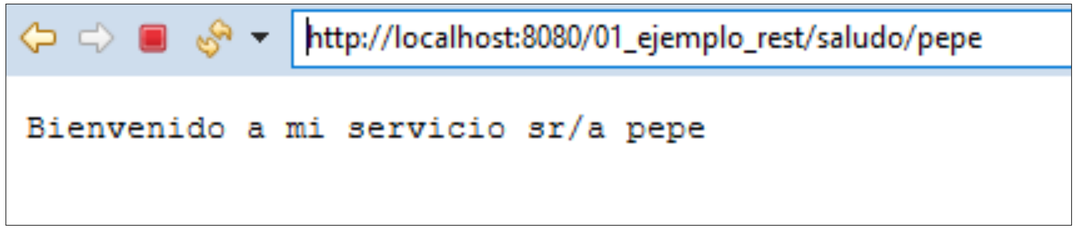
A la hora de recoger estos datos en el método correspondiente, deben declararse las variables en el atributo *value* de la anotación, indicando los nombres de las mismas entre llaves. Además, se **asociarán a parámetros del método mediante la anotación @PathVariable**. Por ejemplo, si queremos incluir en nuestro servicio de ejemplo un recurso que nos mande un saludo personalizado para el nombre que se envía como variable en la URL de llamada, así quedaría la definición del nuevo método:

```
@GetMapping(value="saludo/{nombre}",produces=MediaType.TEXT_PLAIN_VALUE)
public String saludoPersonal(@PathVariable("nombre") String name) {
   return "Bienvenido a mi servicio sr/a "+name;
}
```

Si lanzamos la siguiente URL:

`http://localhost:8080/01_ejemplo_rest/saludo/pepe`

Obtendremos el resultado indicado en la imagen:



Otros métodos HTTP

En los ejemplos que hemos visto hasta el momento sobre creación de servicios se ha trabajado solamente con peticiones HTTP de tipo GET, pero en la implementación de servicios REST se suelen crear recursos que atiendan a otros tipos de peticiones, como **POST (para inserción de datos), PUT(actualización de datos) y DELETE(eliminación)**.

A continuación, vamos a implementar un nuevo servicio que nos permita gestionar una hipotética base de datos de contactos. Cada contacto tendrá un dni, nombre e email y, para simplificar el proceso, en lugar de disponer de una base de datos real almacenaremos los contactos en memoria dentro de la clase del servicio.

Lo primer que haremos será crear un proyecto Web dinámico, como en el ejemplo anterior, y añadir las mismas **dependencias Maven**

En los siguientes listados, vemos como sería la clase *Contacto* y la estructura inicial de la clase controladora, seguidamente iremos implementando los métodos del servicio:

```
public class Contacto {
    private String dni;
    private String nombre;
    private String email;
    public Contacto(String dni, String nombre, String email) {
        super();
        this.dni = dni;
        this.nombre = nombre;
        this.email = email;
    }
//setter y getter
}
```

```
@RestController
public class ContactosController {
    private List<Contacto> contactos;
    @PostConstruct
    public void init() {
        contactos=new ArrayList<>();
    }
    //métodos para peticiones HTTP
}
```

Otros métodos HTTP

Peticiones GET

Lo primero será exponer los recursos asociados a las ya conocidas peticiones GET. Uno de ellos nos devolverá la lista de contactos existentes y otro el contacto cuyo *dni* se reciba como parámetro en URL:

```
@GetMapping(value="contactos",produces=MediaType.APPLICATION_JSON_VALUE)
public List<Contacto> recuperarContactos(){
   return contactos;
}
@GetMapping(value="contactos/{dni}",produces=MediaType.APPLICATION_JSON_VALUE)
public Contacto buscarContacto(@PathVariable("dni") String dni ){
   for(Contacto con:contactos) {
      if(con.getDni().equals(dni)) {
         return con;
      }
   }
   return null;
}
```

En ambos casos, **el tipo producido es JSON**, aunque le primer método devolverá un array de objetos mientras que el segundo solamente un único objeto. También vemos que ambos métodos tienen asociada la misma URL, diferenciándose por la variable en URL.

Dejaremos el tema de las pruebas para el final, una vez que hayamos implementado el servicio al completo

Otros métodos HTTP

Peticiones POST

Seguidamente, vamos a implementar un recurso que realice la operación de añadir contacto a la lista. Este recurso lo implementaremos en un método asociado a una petición de tipo POST. Estas peticiones permiten enviar datos en el cuerpo de la respuesta, en nuestro caso, un objeto JSON con los datos del contacto.

A través de la anotación **@RequestBody** y el atributo *consumes* de la anotación **@PostMapping**, le indicaremos a Spring que **transforme el objeto JSON recibido en el cuerpo en un objeto Java de tipo Contacto.**

```
@PostMapping(value="contactos",consumes=MediaType.APPLICATION_JSON_VALUE)
public void nuevoContacto(@RequestBody Contacto contacto) {
   contactos.add(contacto);
}
```

Podemos observar como la URL indicada en el método es la misma que en el método de recuperación de contactos, pero al estar asociado a una petición de tipo POST **no habrá problema de ambigüedad** al llegar una petición con dicha URL.

También vemos que, en este caso, el método es de tipo *void* y no devuelve ningún resultado, lo cual no significa que tenga que ser siempre así en métodos asociados a peticiones POST.

Otros métodos HTTP

Peticiones PUT

El siguiente recurso implementará una operación de actualización. El objeto JSON con los datos del contacto recibido en el cuerpo, sustituirá al objeto con mismo DNI que exista en la base de datos.

Al igual que en POST, usamos la anotación **@RequestBody** y el atributo *consumes* de la anotación **@PutMapping**, para el mapeo del objeto JSON que llega en el cuerpo a un objeto Contacto:

```
@PutMapping(value="contactos",consumes=MediaType.APPLICATION_JSON_VALUE)
public void actualizarContacto(@RequestBody Contacto contacto) {
   for(int i=0;i<contactos.size();i++) {
      //si coinciden los dni, se realiza una sustitución
      if(contactos.get(i).getDni().equals(contacto.getDni())) {
         contactos.set(i, contacto);
      }
   }
}
```

Otros métodos HTTP

Peticiones DELETE

El último recurso realizará la eliminación de un contacto de la base de datos, cuyo *dni* coincida con el recibido en la URL.

En este ejemplo, hemos optado por que el método devuelva el contacto eliminado:

```
@DeleteMapping(value="contactos/{dni}",produces=MediaType.APPLICATION_JSON_VALUE)
   public Contacto eliminarContacto(@PathVariable("dni") String dni ){
      //recorre los contactos y elimina de la base de datos
      //el contacto que coincida con el dni
      for(int i=0;i<contactos.size();i++) {
         if(contactos.get(i).getDni().equals(dni)) {
            Contacto con=contactos.get(i);
            contactos.remove(i);
            return con;
         }
      }
      return null;
   }
```

Aplicación Postman

Desde un navegador solo podemos lanzar peticiones GET, por lo que si queremos probar los recursos asociados a peticiones PUT,POST y DELETE, tendremos que utilizar otra herramienta.

Esta herramienta es **Postman**, una aplicación que permite lanzar de forma sencilla todo tipo de peticiones HTTP, incluir parámetros en la misma y ver los resultados obtenidos.

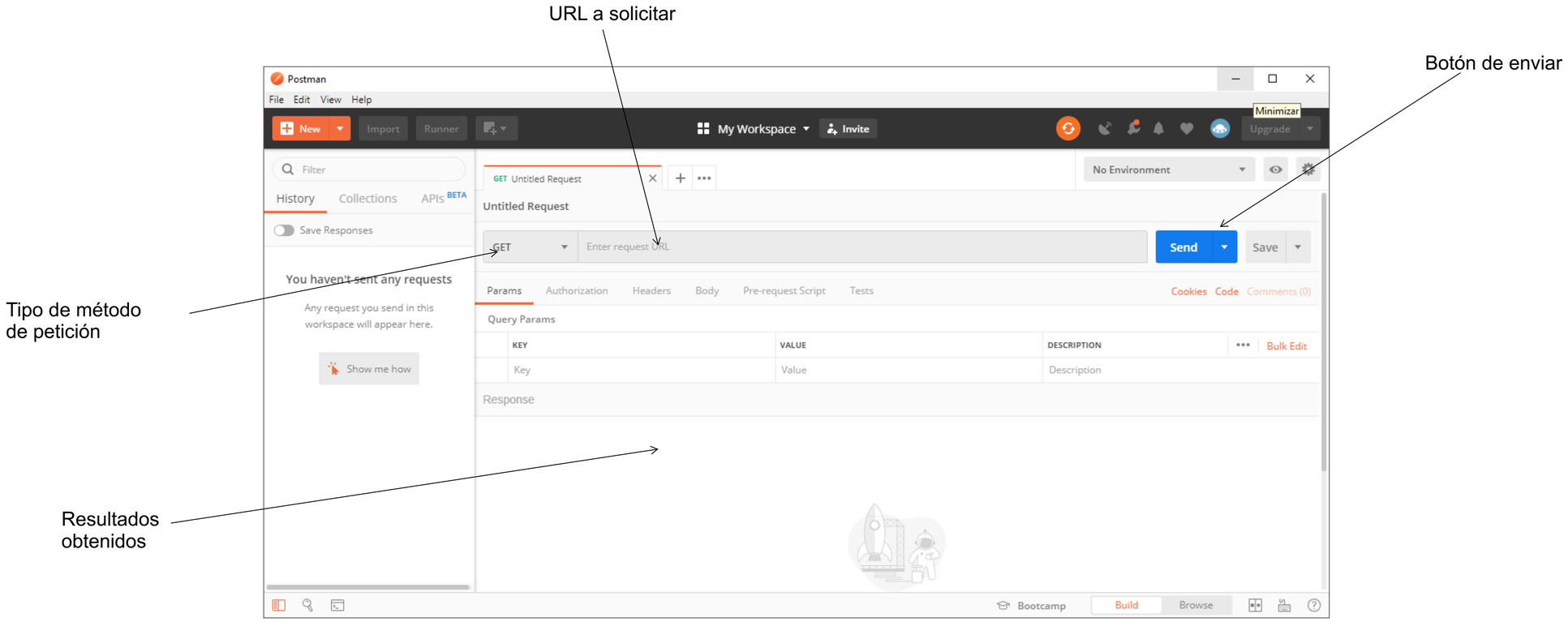
Podemos descarga Postman desde la siguiente URL:

https://www.getpostman.com/downloads/

Una vez descargado, el proceso de instalación es sencillo, no tenemos más que aceptar todas las opciones hasta el final.

Aplicación Postman

Al iniciar postman por primera vez, su aspecto será similar al que se indica a continuación:

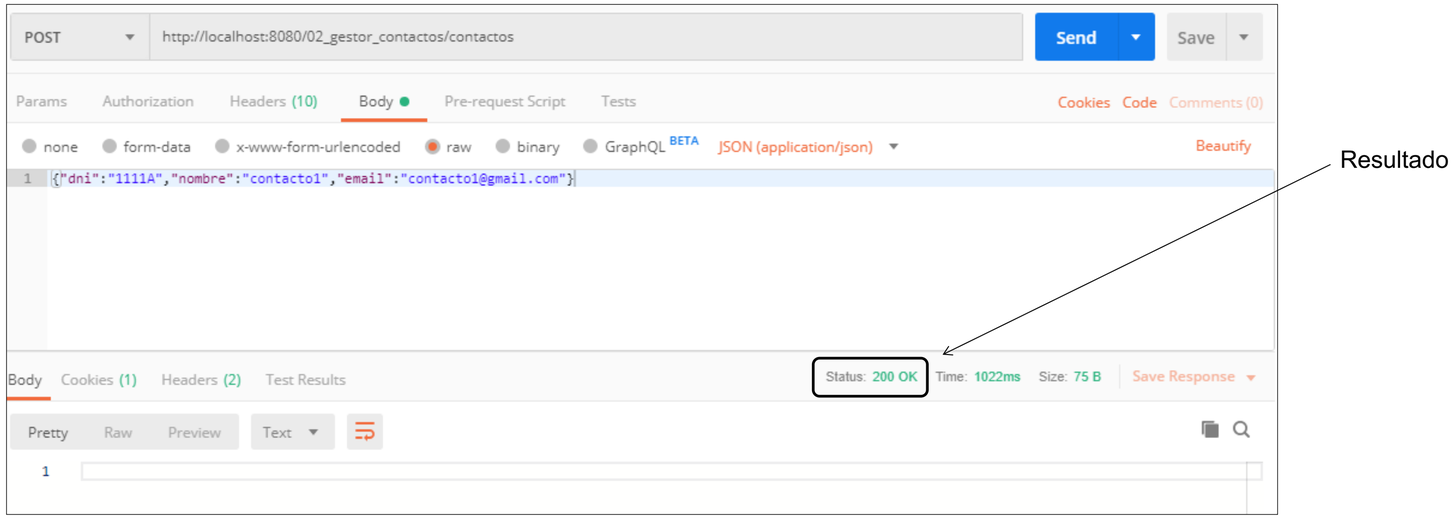


Aplicación Postman

Petición POST

Una vez iniciado el servicio de contactos, utilizaremos Postman para lanzar diferentes peticiones al mismo y probar su funcionamiento. Empezaremos con un par de peticiones tipo POST para añadir dos contactos al conjunto. Para ello, seleccionaremos esa opción en el tipo de método y escribiremos la dirección http://localhost:8080/02_gestor_contactos/contactos en el campo de texto de dirección.

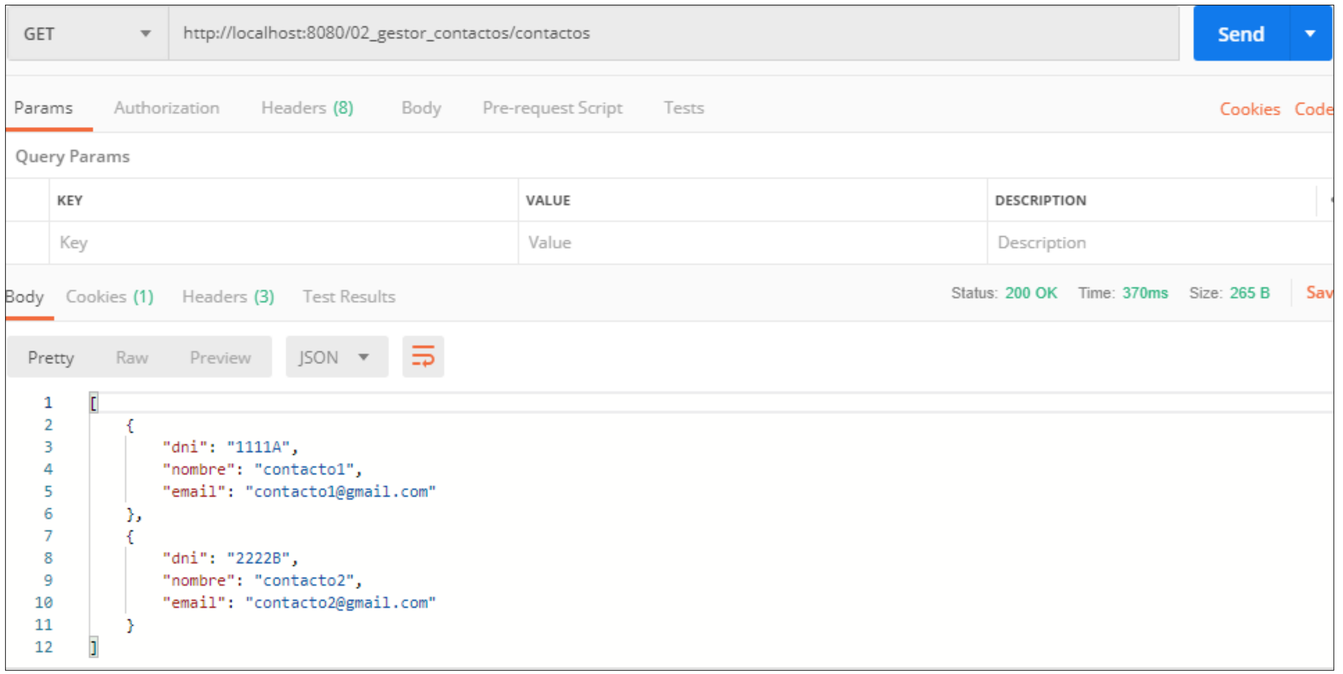
En las pestañas que aparecen debajo de la dirección, seleccionaremos “Body” y marcaremos la opción “raw”, eligiendo además la opción JONS (application/json) en la lista que aparece a la derecha. Para terminar, **introducimos un documento JSON con los datos del contacto que queremos añadir** y pulsamos el botón *Send* para lanzar la petición. Si la respuesta es 200 ok, entonces es que se ha realizado correctamente la petición, por lo que procedemos a añadir un segundo contacto.



Aplicación Postman

Petición GET

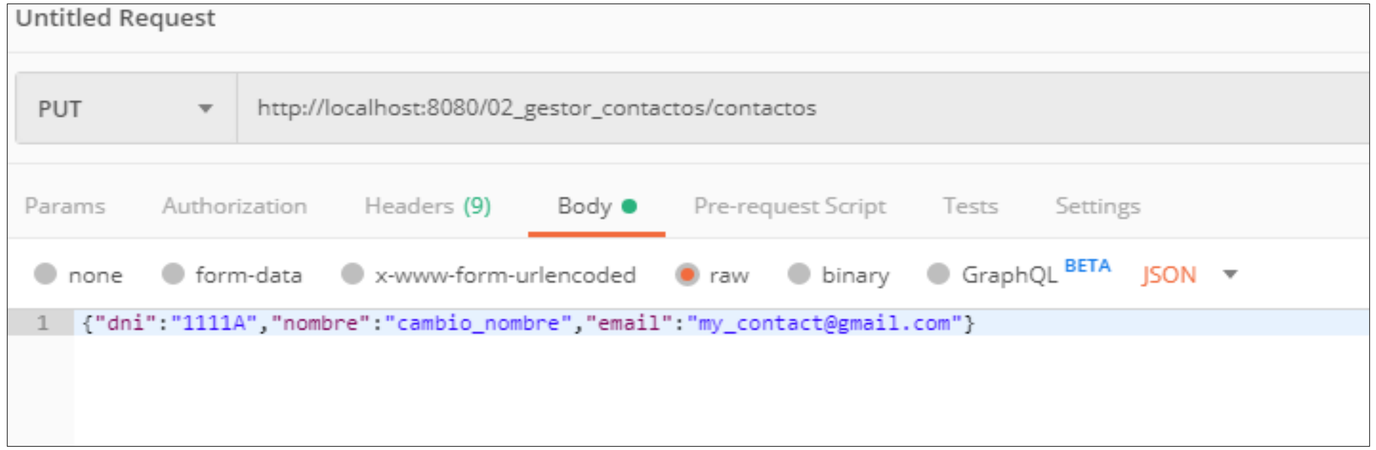
Seguidamente lanzaremos una petición GET para comprobar si los datos se han añadido correctamente y también si se devuelven en de la forma esperada. En una pestaña nueva (para no perder lo anterior) introduciremos la misma dirección de antes y pulsaremos el botón **Send**. Si todo ha ido bien, veremos en la **sección de respuesta un array de objetos JSON** con los dos contactos agregados anteriormente.



Aplicación Postman

Petición PUT

Aunque no necesariamente, una petición PUT puede incluir un documento JSON en el cuerpo de la misma con los datos a actualizar, tal y como ocurre en el ejemplo que hemos desarrollado. En este caso, el proceso para lanzarla será similar a las peticiones POST, **enviando un objeto JSON con los datos del contacto a modificar**.

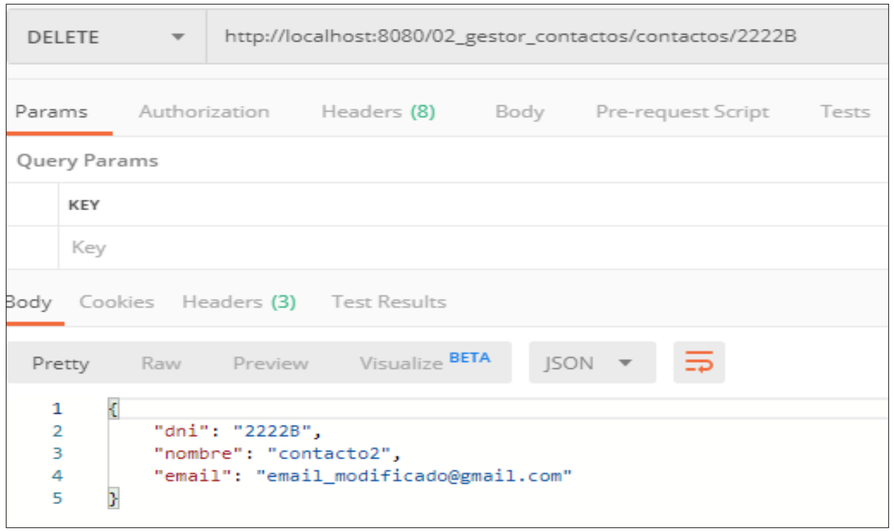


Podemos volver a lanzar después una petición GET para comprobar si la actualización se ha realizado correctamente.

Aplicación Postman

Petición DELETE

Para lanzar una petición de tipo DELETE, simplemente la seleccionaremos en la lista de métodos y en la **url indicaremos el dni del contacto a eliminar**.



Además de que en la respuesta recibiremos el objeto eliminado, si después volvemos a lanzar de nuevo una petición GET comprobaremos que el contacto ya no está.

Ejercicio Práctico

Tenemos una base de datos con una tabla que almacena los cursos de un centro de formación. La tabla tiene los siguientes campos: idCurso (autonumérico), denominacion (texto), duración(entero) y fechaInicio(DateTime).

Se pide implementar un servicio REST que ofrezca los siguientes recursos:

- Devolución de la lista de cursos existentes ante una petición GET.
- Búsqueda de un curso a partir del idCurso enviado en la URL de una petición GET.
- Eliminación de un curso a partir del idCurso enviado en la URL de una petición DELETE.
- Alta de un nuevo curso a partir de los datos del mismo, recibidos en el cuerpo de una petición POST.
- Modificación de la denominación de un curso a partir del idCurso y nuevo nombre, que serán recibidos como variables de URL en una petición PUT.

Solución ejercicio

- En el siguiente video tutorial podrás ver la solución al ejercicio propuesto, con la explicación detallada de los componentes desarrollados.

VER TUTORIAL

Recuerda

- Un servicio REST expone una serie de recursos en la red a los que se accede a través de peticiones HTTP.
- Para crear servicios REST, podemos utilizar los módulos Spring Web y Spring MVC, que proporcionan una serie de anotaciones para indicarle al framework como tiene que comportarse, de modo que el programador se concentre en el desarrollo de la lógica de la aplicación.
- Los servicios Spring REST se crear como aplicaciones Web y se despliegan en un servidor de aplicaciones Java EE. Además de las dependencias Spring Web y MVC, se deben incluir las básicas de Spring core.
- Las aplicaciones Servicios REST deben ser configuradas como cualquier aplicación Web de Spring, incluyendo los archivos web.xml y otro específico de Spring.
- De cara a probar un servicio REST, podemos utilizar la aplicación Postman, la cual nos permite lanzar todo tipo de peticiones HTTP a un servicio, enviar datos en URL y cuerpo y comprobar los resultados devueltos por el mismo.