

# Como crear un microservicio o servicio web REST con Spring Boot

## (Laboratorio 1)

### Contenido

- 1 Diseño del microservicio
- 2 Creando el proyecto
- 3 Agregando la capa DTO
- 4 Creando el API

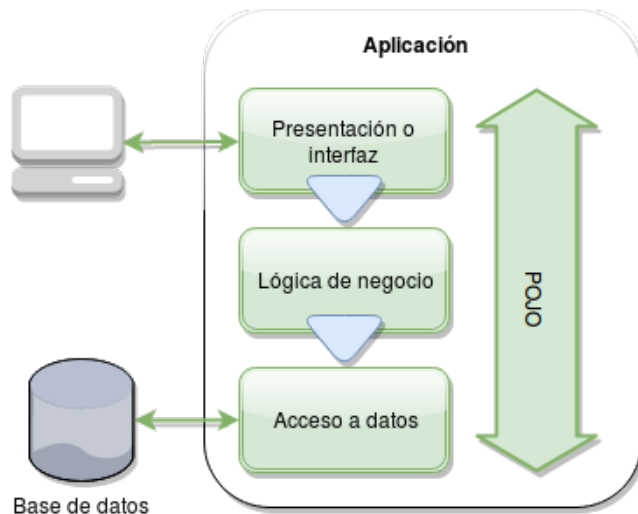
### Diseño del microservicio

Para utilizar alguna temática, crearemos un servicio web para la administración de información de contactos, como nombre, teléfono, email, etc.

Una buena práctica de diseño de software (debería ser obligatoria!) es la aplicación del principio de separación de responsabilidades (*separation of concerns*) que consiste en asegurarse que cada clase, componente, etc. tenga una responsabilidad bien definida, es decir, que sus funcionalidades o métodos no hagan más de un tipo de actividad, por ejemplo, dibujar la página web y guardar datos en una base datos.

Un ejemplo de la falta de separación de responsabilidades es cuando en un script PHP se “pintan” tags HTML y unas (miles!) líneas más abajo, se establece la conexión a la base de datos para consultar o guardar información. Esto lleva a producir software difícil de mantener, actualizar y corregir, entre muchos otros problemas muy graves.

La separación de responsabilidades se aplica en diferentes formas y niveles. En este caso la usaremos para separar tres responsabilidades bien definidas:



1. Interactuar con el cliente del servicio (capa interfaz o presentación). Aquí agrupamos las clases y métodos que conforman el **API** del servicio web.
2. Ejecutar procesamientos y aplicar reglas de negocio. Estas son clases que tienen la “sabiduría” o conocimiento sobre la lógica de lo que hace el servicio web. Algunos llaman a estas clases, “**servicios**”
3. Acceso a datos. Son las clases que se encargan de almacenar y extraer la información de un repositorio. Algunos las llaman *Repository* o *Data Access Object (DAO)*

Adicionalmente tenemos un tipo de objeto que transporta la información entre estas capas de nuestra aplicación, para reducir el acoplamiento. Son los llamados *Data Transport Object (DTO)* o *Value Object (VO)* que se implementan como objetos planos de java (*Plain Old Java Object* o **POJO**), es decir, simples contenedores de datos, sin métodos o comportamiento.

Un tipo de objeto muy parecido a los DTO son las entidades (**entity**) que representan tablas o registros en la base de datos. La mejor práctica es extraer un entity de la base de datos y asignar algunos de sus valores a un DTO equivalente para retornarlo a la capa superior en lugar de retornar el entity. Es una práctica discutible porque por ser tan parecidas las entities con los DTO, el código parecerá duplicado; en su favor argumentamos que evita que el cliente reciba información muy detallada de la base de datos o campos que no queremos que reciba (ej: hash del password), reduciendo así el acoplamiento y mejorando la abstracción.

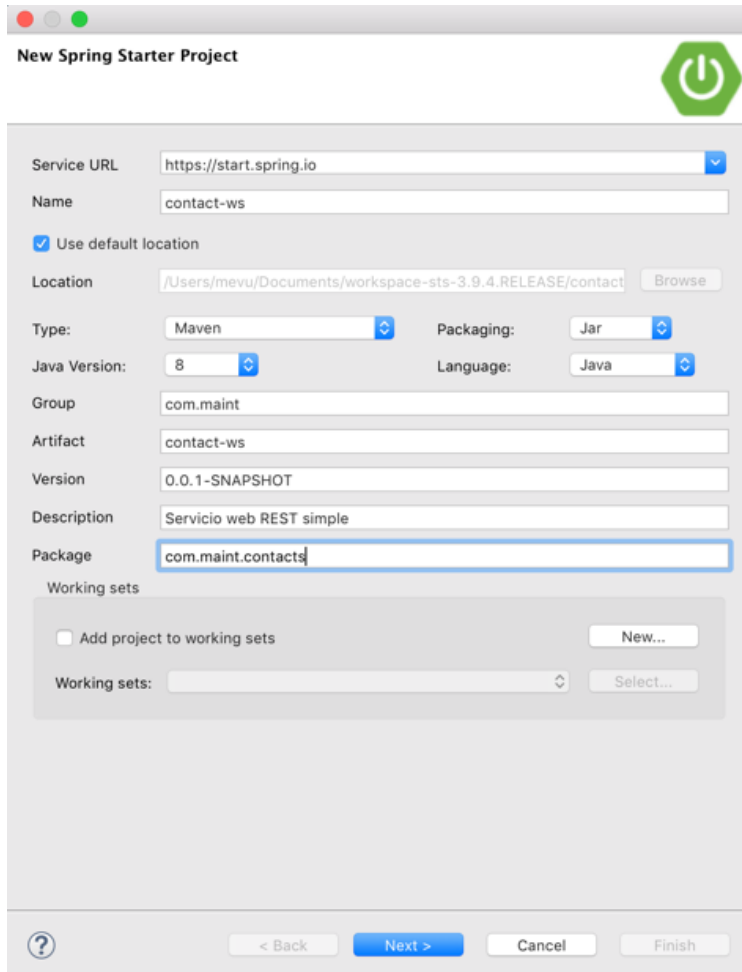
## Creando el proyecto

Usaremos un proyecto Maven que es genérico y puede usarse en eclipse, Netbeans, IntelliJ IDEA o cualquier otro IDE Java, incluso con editores de texto de línea de comandos.

Tenemos por lo menos tres formas de crear el proyecto:

1. Por comandos de maven: simple pero entonces este post sería sobre Maven y no sobre Spring Boot...
2. Usando el sitio web de Spring Boot Initializr, en el cual podremos seleccionar los parámetros del proyecto, las librerías a incluir y descargar un zip que contiene nuestro proyecto base, el cual debemos importar en nuestro IDE favorito
3. Utilizando Spring Tool Suite (STS) que es un conjunto de plugins para Eclipse, especialmente diseñados para hacernos muy productivos al desarrollar una aplicación de Spring. Usaremos esta opción que básicamente es una fachada pues STS se conecta a Spring Boot Initializr para hacer lo mismo que haríamos visitando el sitio web, sólo que importa automáticamente el proyecto en nuestro IDE.

Para crear el proyecto, en STS selecciona el menú *"File/New/Spring Starter Project..."* y en la ventana del asistente, ingresa los datos de tu proyecto, por ejemplo en nuestro caso, se verían así:



**New Spring Starter Project**

Service URL:

Name:

☒ Use default location

Location:

Type:  Packaging:

Java Version:  Language:

Group:

Artifact:

Version:

Description:

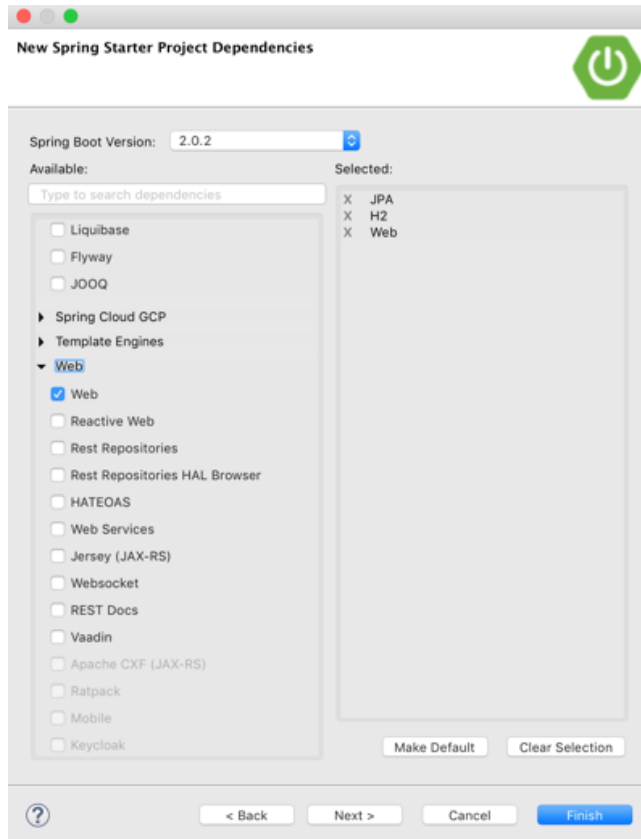
Package:

Working sets

☐ Add project to working sets

Working sets:

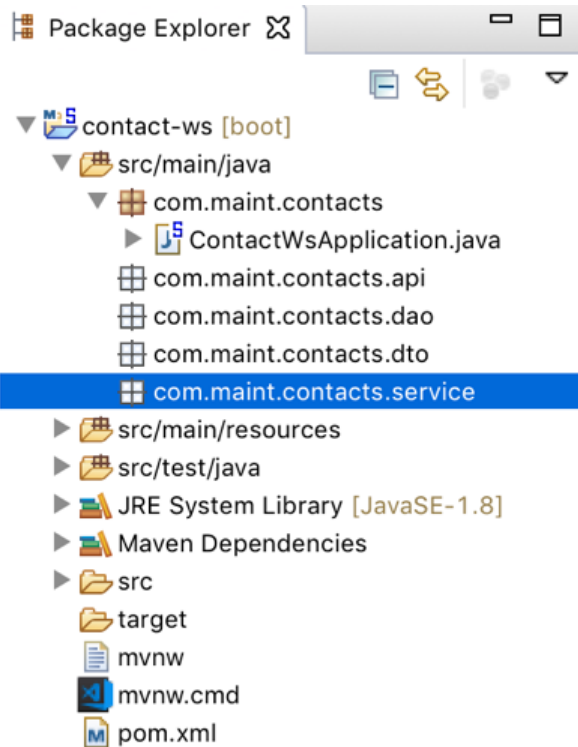
En la siguiente ventana selecciona las librerías **Web** (clases para páginas y servicios web), **JPA** (Hibernate y otras librerías para acceder a la base de datos) y **H2** (base de datos en memoria para facilitar el desarrollo y pruebas). Presiona el botón “Finish” para completar la generación del proyecto:



Unos pocos segundos o minutos después, tendrás el proyecto listo en tu IDE. Pruébalo ejecutándolo como una aplicación Java normal: selecciona el proyecto, click derecho y pulsa el menú *“Run as.../Java Application”*. También funciona con el menú *“Run as.../Spring Boot Application”*. Con este último, el log de la consola se verá en colores facilitando su lectura. Si todo va bien, el proyecto debería iniciar sin errores en el log.

## Agregando la capa DTO

Vamos a crear una clase simple (POJO) para representar a un contacto. Una buena práctica es agrupar las clases en paquetes según su responsabilidad: **dao**, **dto**, **service**, **api**, etc., así que crea estos paquetes (o como los quieras llamar), así:



En el paquete dto o entity o como hayas llamado al que contendrá tus objetos de datos, crea la clase **Contact** y añade campos como un id único, nombre, apellido, teléfono, email y otros que consideres:

```
1 package com.maint.contacts.dto;
2
3 public class Contact {
4     Long id;
5     String firstName;
6     String lastName;
7     String phoneNumber;
8     String email;
9 }
```

TIP: Después de agregar los campos básicos, utiliza los menús (con click derecho) “Source/Generate Getters and Setters...”, “Source/Generate Constructor using Fields...” y “Source/Generate toString()...” para generar el resto del código requerido por el POJO o DTO. toString() te será muy útil en la depuración y registro en el log y para mejor rendimiento, utiliza StringBuffer o StringBuilder.

## Creando el API

Para crear la interfaz REST del servicio web, crea la clase `ContactsApi` en el paquete `Api` y agrega un método básico para consultar un contacto por un ID, así:

```
1 package com.maint.contacts.api;
2
3 import com.maint.contacts.dto.Contact;
4
5 public class ContactsApi {
6
7     public Contact getByld(){
8         return new Contact(1L, "John", "Doe", "+57 311 222 3344", "john@maint.com");
9     }
10 }
```

Por ahora el contacto estará “quemado” (*hard-coded*) en el API pero más adelante cambiaremos esto para que lo consulte de la base de datos.

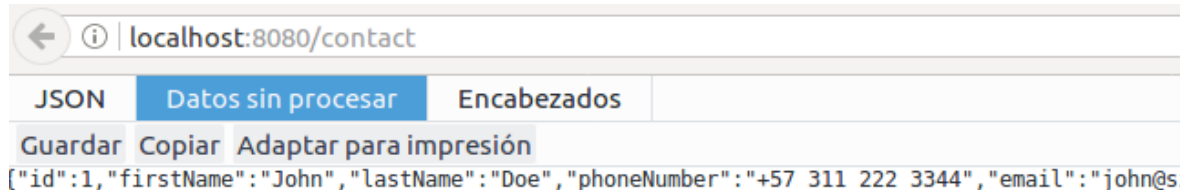
Ya tenemos nuestra clase `Api` pero aún le falta para que sea un servicio web: para que Spring reconozca una clase como servicio web y la trate como tal, debemos anotar la clase con **@RestController**. También debemos anotar cada método público que deseemos

exponer a los clientes con la anotación **@RequestMapping**, así:

```
1 @RestController
2 public class ContactsApi {
3
4     @RequestMapping(value="/product", method=RequestMethod.GET)
5     public Contact getByld(){
6         return new Contact(1L, "John", "Doe", "+57 311 222 3344", "john@maint.com");
7     }
8 }
```

La anotación **@RequestMapping** recibe parámetros que nos ayudan a cambiar la definición del método hacia el exterior: los content type que recibe o entrega, los headers que requiere, etc. En este caso los básicos son **value**, para indicar la url en la cual “escuchará” este método y **method** que indica el método HTTP que iniciará la ejecución de esta operación. Por defecto es GET, así que en este caso podríamos no dejar sólo el parámetro **value** y el servicio funcionaría igual.

Ahora ejecuta la aplicación y dirige el navegador hacia `http://localhost:8080/product` (si no has cambiado las configuraciones por defecto) y verás algo similar a esto, según tu navegador:





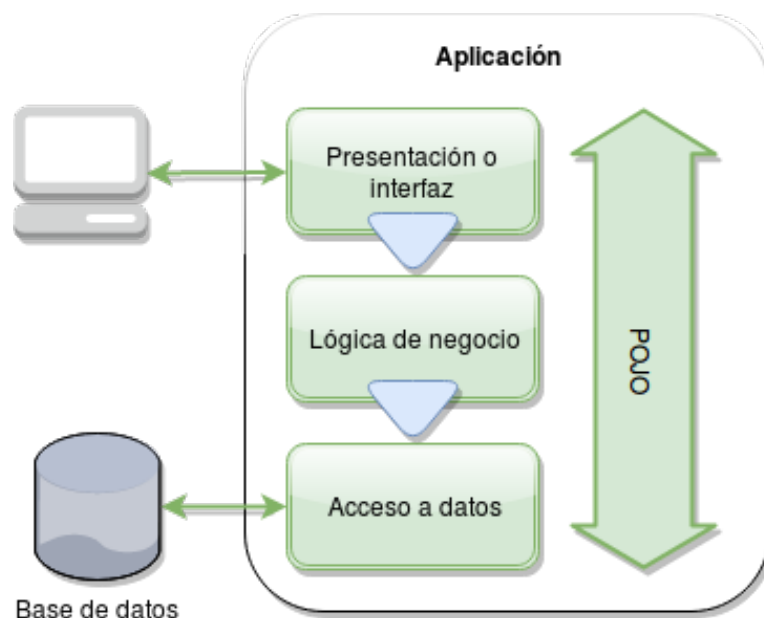
## Como crear un microservicio o servicio web REST con Spring Boot (Laboratorio 2)

### Contenido

- 1 Creando el modelo de datos
- 2 Capa de acceso a datos
- 3 Capa de lógica
- 4 Usando la lógica desde la interfaz
- 5 Advertencia!

En este segundo laboratorio agregaremos el acceso a datos mediante consultas y actualizaciones a una base de datos relacional. En la parte final lo enriqueceremos con algunas de las características que hacen tan poderoso a Spring Framework y corregiremos fallas de diseño que ya tiene nuestro servicio.

Como recordarás, en la primera parte de este post dividimos la aplicación en las tres capas lógicas del siguiente diagrama y construimos la capa de interfaz.



En este laboratorio construiremos la capa de lógica de negocio y la de acceso a datos.

## Creando el modelo de datos

Spring Boot nos ofrece múltiples formas de almacenar nuestros datos mediante un gran número de integraciones con repositorios populares como bases de datos no relacionales (MongoDB, Cassandra, etc), directorios LDAP y bases de datos relacionales tradicionales (MySQL, PostgreSQL, Oracle, etc) utilizando JDBC y la especificación JPA.

Si terminaste el primer laboratorio, ya tienes incluidas las librerías de JPA, Hibernate y el driver JDBC para la base de datos (relacional) H2 en tu archivo pom.xml, lo que resta es crear la entidad que representará a una tabla de la base de datos. Para esto utilizaremos únicamente anotaciones estándar de JPA y usaremos nuestra clase Contact para convertirla

en un entidad, así:

```
1 // (1)
2 import java.io.Serializable;
3
4 import javax.persistence.Entity;
5 import javax.persistence.GeneratedValue;
6 import javax.persistence.Id;
7
8 // (2)
9 @Entity
10 public class Contact implements Serializable {
11
12     // (3)
13     private static final long serialVersionUID = 4894729030347835498L;
14
15     // (4)
16     @Id
17     @GeneratedValue
18     private Long id;
19     private String firstName;
20     private String lastName;
21     private String phoneNumber;
22     private String email;
23     ...
```

1. Imports nuevos para agregar anotaciones de JPA

2. Indicamos que esta clase representa a una tabla de base de datos con el mismo nombre. Para cambiar el nombre de la tabla podemos usar la anotación `@Table`
3. Es buena práctica hacer las entidades serializables. Algunos proveedores de persistencia (Hibernate, EclipseLink, etc.) pueden presentar excepciones si en algunos casos particulares la entidad no es serializable.
4. La anotación `@Id` indica que este atributo será la clave primaria y `@GeneratedValue` indica la forma en que se generarán los valores de la clave primaria. En este caso se usará el valor por defecto que hace que se use el valor que genere la base de datos. El comportamiento se puede cambiar por ejemplo, a una secuencia de Oracle, usando el parámetro “strategy”

## Capa de acceso a datos

Spring Data JPA, que ya está entre nuestras dependencias, facilita mucho la creación de clases que guarden y recuperen datos de la base de datos usando JPA. Tradicionalmente debíamos crear manualmente la conexión a base de datos (datasource), configurar entity manager/unidad de persistencia y crear todo el código CRUD en una clase auxiliar. Este es un trabajo repetitivo y propenso a errores por lo que Spring Data JPA se encarga de estas configuraciones automáticamente, atendiendo convenciones que podemos cambiar por configuración, de llegar a requerirlo.

La forma más rápida y segura de crear una clase de acceso a datos es usar la interfaz `JpaRepository` de Spring Data JPA. Para esto, crea una interfaz vacía en el paquete dao que la extienda, así:

```
1package com.maint.contacts.dao;
2
3import org.springframework.data.jpa.repository.JpaRepository;
4
5import com.maint.contacts.dto.Contact;
```

```
6
7public interface ContactRepository extends JpaRepository<Contact, Long> {
8
9}
```

Fíjate que esta interfaz extiende la interfaz `JpaRepository`. Durante la inicialización de la aplicación, Spring Data busca estas interfaces y crea clases que las implementan, ofreciendo automáticamente los métodos típicos para Crear, Actualizar o Eliminar (CRUD) una entidad. Los parámetros de esta interfaz son: la entidad a gestionar (`Contact`) y el tipo de dato de su clave primaria (`Long`).

## Capa de lógica

En este caso esta capa es muy simple pues no requerimos hacer operaciones con reglas de negocio o cálculos complejos y por ahora, sólo vamos a guardar los datos tal como nos llegan. En un servicio más complejo, es aquí en donde debemos dejar todas las operaciones que implementen la lógica de negocio, por ejemplo, consultar servicios externos, hacer validaciones complejas, calcular otros campos, etc., haciendo que nuestro `@RestController` tenga la menor cantidad de lógica posible.

Crea la clase “`ContactService`” en el paquete “`service`” así:

```
1 // (1)
2 @Service
3 public class ContactService {
4     // (2)
5     @Autowired
6     ContactRepository dao;
7
8     // (3)
9     public Contact save(Contact contact){
10         return dao.saveAndFlush(contact);
11     }
12 }
```

1. La anotación o estereotipo “@Service” indica a Spring que cree una instancia de esta clase (bean) que podremos usar en otras instancias
2. @Autowired es equivalente a @Inject que se usa en ambientes JavaEE, incluso esta última funciona con Spring también. Se utiliza para indicar a Spring que queremos que asigne una instancia de un bean a la variable que tiene la anotación @Autowired
3. Nuestro método de lógica de negocio. En este caso la lógica es mínima pero es en este tipo de métodos y clases en donde debemos concentrar las operaciones, condiciones, reglas y demás acciones de lógica de negocio de nuestra aplicación

## Usando la lógica desde la interfaz

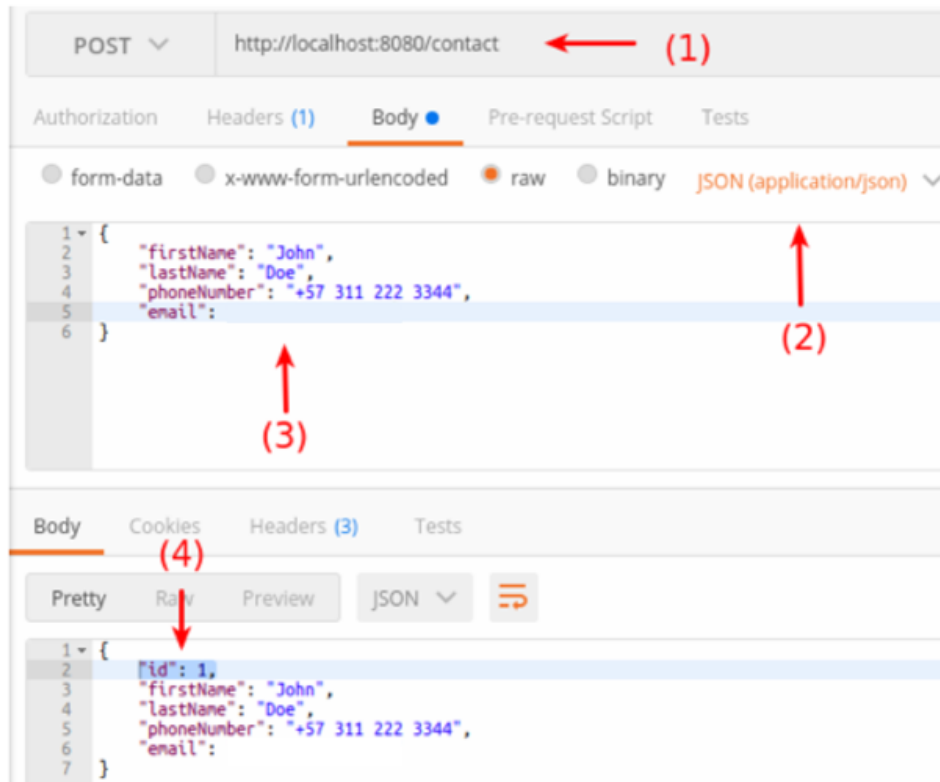
¿Ya tenemos la lógica de negocio, como la usamos desde nuestra capa de interfaz?

Debemos modificar la clase ContactsApi para que haga uso de nuestra clase de lógica de negocio, así:

```
1 ...
2 // (1)
3 @Autowired
4 ContactService contactService;
5
6 // (2)
7 @RequestMapping(value="/contact", method=RequestMethod.POST)
8 public Contact updateOrSave(@RequestBody Contact contact){
9     // (3)
10    return contactService.save(contact);
11}
12...
```

1. Indicamos a Spring que debe inyectar una instancia de nuestra clase de lógica de negocio
2. Creamos un método REST que mediante una petición HTTP POST en la url `http://localhost:8080/contact`. Fíjate en la anotación @RequestBody antes de la variable “contact”. Esta anotación indica que la variable debe ser creada con los valores que lleguen en el cuerpo de la petición HTTP, para lo cual usaremos una estructura JSON
3. Invocamos lógica de negocio contenida en nuestra clase ContactService

Para hacer la petición POST podemos utilizar herramientas como SoapUI o Postman. En la siguiente imagen se muestra una petición utilizando Postman (3, en la parte superior) y la respuesta del servicio (4, en la parte inferior):



1. Url de nuestro servicio web y método POST seleccionado
2. Encabezado de la petición que indica el tipo de petición (Content-Type: application/json)
3. Cuerpo de la petición. Es una estructura JSON que representa un contacto. Observa que no se ha indicado el valor del campo Id ya que este campo es generado por la base de datos, según las anotaciones de nuestra entidad
4. Respuesta del servicio. Observa que el servicio nos retorna la entidad con un valor numérico autogenerado en el campo Id

## Advertencia!

Ya tienes un servicio web REST básico funcionando, integrado con una base de datos relacional en memoria.. Lo hemos hecho de este modo para facilitar la comprensión de los conceptos básicos a los principiantes, sin embargo **este servicio no está listo para ser usado en una aplicación real porque tiene fallas de diseño** que dificultarán su evolución, las pruebas automatizadas y van en contra del bajo acoplamiento que debe tener un buen diseño. Corregiremos estos problemas en nuestro próximo laboratorio.

## Como crear un microservicio o servicio web REST con Spring Boot (Laboratorio 3)

### Contenido

- 1 Desacoplando a los clientes
  - 1.1 El modelo de datos del API
  - 1.2 Mapeando entre modelos de datos
- 2 Validación automática de datos de entrada
- 3 Conectando con una base de datos externas
  - 3.1 Perfiles y Ambientes
- 4 Conclusión

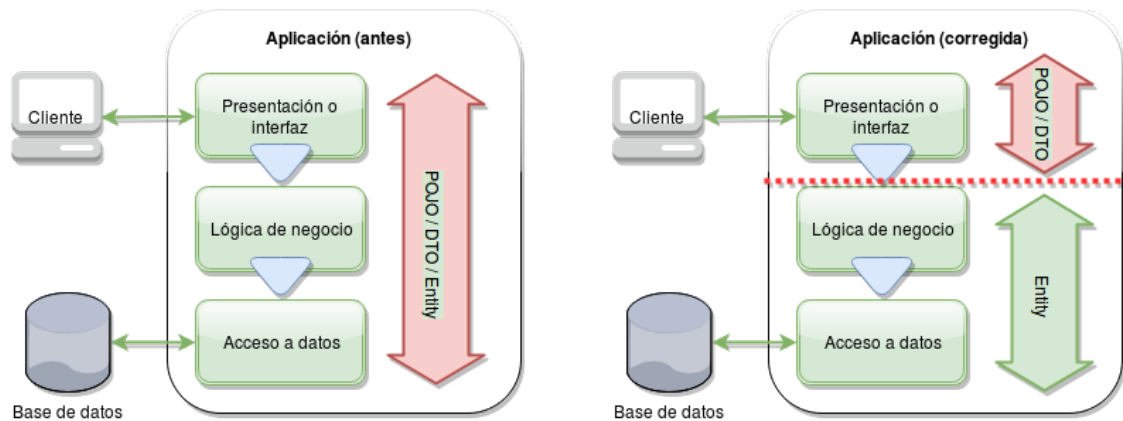
En el primer laboratorio creamos la interfaz de un microservicio o servicio web REST usando Spring Boot y en el segundo laboratorio agregamos la lógica de negocio y el acceso a datos, sin embargo el servicio tiene problemas de diseño que causan acoplamientos negativos, dificultan su evolución y las pruebas automatizadas. En esta último laboratorio corregiremos esos problemas y haremos uso de varias características avanzadas de Spring Boot y otras librerías.

### Desacoplando a los clientes

La interfaz de servicio (*ContactsApi*) retorna objetos de tipo *Contact* que hemos marcado como entidades con la anotación *@Entity*; en otras palabras: nuestro servicio web está exponiendo a sus clientes, el modelo de datos tal como es. Esto implica que si hacemos un cambio en la estructura de base de datos, el cambio automáticamente se verá en la interfaz de servicio y afectará a los clientes del servicio web, por lo que decimos que existe un fuerte *acoplamiento negativo* entre los clientes y la interfaz del servicio web. Este es un antipatrón también conocido como *Modelo del Dominio como REST*. La forma de solucionarlo, ilustrada en el siguiente diagrama, es crear objetos independientes para la



interfaz de servicio y objetos (entidades) diferentes para el modelo de datos; de esta manera podremos hacer cambios en el modelo de datos sin afectar obligatoriamente a los clientes.



Lo malo de este enfoque es que tendremos que hacer un mapeo entre los campos de la entidad y los campos de los objetos que usamos en la interfaz de servicio web. Hay varias formas en la que podemos hacer este mapeo:

1. Manualmente, escribiendo una clase auxiliar que haga el mapeo para que sea reutilizable
2. Como parte de la lógica de los POJOs/DTOs de la capa de interfaz
3. Utilizando una librería de mapeo de objetos como Dozer, ModelMapper o MapStruct .

Como este último laboratorio, usaremos Dozer.

## El modelo de datos del API

Crea las clases *ContactRequest* y *ContactResponse*. Estas clases serán los POJOs que el servicio web recibirá y entregará a sus clientes, en lugar de la entidad *Contact*, por lo que su estructura es muy similar y puedes verla en los fuentes que están en nuestro repositorio en GitHub. Al evitar que la interfaz del servicio web entregue a sus clientes cualquiera de

nuestras entidades, ya los hemos desacoplado de nuestro modelo de datos del servicio que es diferente al modelo de datos del API.

Este cambio implica que debemos cambiar la interfaz del servicio web (Contacts) para que reciba y entregue instancias de estas nuevas clases, así:

```
1 public ContactResponse updateOrSave(@RequestBody ContactRequest contactRequest)
```

## Mapeando entre modelos de datos

También debemos agregar los mapeos correspondientes entre los DTO que se intercambian con el cliente (ContactResponse y ContactRequest) y las entidades (Contact) que recibe y entiende nuestra clase de lógica de negocio (ContactService). Para esto debemos agregar la librería Dozer al archivo pom.xml y configurar un bean de Dozer en una clase de configuración:

```
1<dependency>
2 <groupId>net.sf.dozer</groupId>
3 <artifactId>dozer</artifactId>
4 <version>5.5.1</version>
5</dependency>
1@Configuration // (1)
2public class DozerMapper {
3  @Bean // (2)
4  public Mapper beanMapper() {
5    return new DozerBeanMapper(); // (3)
6  }
7}
```

1. La anotación *@Configuration* indica a Spring que esta clase tiene métodos anotados con *@Bean* que puede procesar en tiempo de ejecución para producir beans que pueden ser inyectados por otros beans
2. La anotación *@Bean* en un método indica que el valor que retorne el método quedará disponible como un bean en el contexto de Spring,, listo para ser inyectado en otro bean

3. Creamos una instancia de DozerBeanMapper que podremos inyectar en nuestro RestController u otro bean de Spring. Por defecto, el nombre del nuevo bean será el mismo nombre del método que lo produce. El nombre se puede personalizar con el parámetro *name*.

El mapper de Dozer intentará asignar los valores de las propiedades de un bean a las propiedades del mismo nombre en otro bean encargándose automáticamente de la conversión de tipos. Dozer también permite mapeos avanzados o personalizados si los nombres de las propiedades no son los mismos (deja un comentario si deseas un tutorial sobre Dozer).

En nuestro caso las propiedades del DTO y del Entity se llaman igual, así que no requerimos especificar mapeos explícitamente y nuestra clase de interfaz (ContactsApi) queda así:

```
1 // Inyecta mapper de Dozer
2 @Autowired
3 Mapper mapper;
4
5 @RequestMapping(value="/contact", method=RequestMethod.POST)
6 public ContactResponse updateOrSave(@RequestBody ContactRequest contactRequest){
7     // Mapeo request dto ==&amp;amp;amp; entity
8     Contact contact = mapper.map(contactRequest, Contact.class);
9
10    // Invoca lógica de negocio
11    Contact updatedContact = contactService.save(contact);
12
13    // Mapeo entity ==&amp;amp;amp; response dto
14    ContactResponse contactResponse = mapper.map(updatedContact, ContactResponse.class);
15
16    return contactResponse;
17 }
```

Este código podría ser más breve pero deseo resaltar cada paso de manera independiente. Con esto hemos desacoplado los clientes de nuestro servicio de nuestro modelo de datos permitiendo su evolución independiente. Veamos ahora otras características de Spring Boot que facilitan el desarrollo de microservicios.

## Validación automática de datos de entrada

Probablemente recuerdas de otros proyectos, algún controller con muchas instrucciones *if* para verificar que los datos de entrada no sean nulos, tengan un valor numérico en un rango específico, tengan una longitud de cadena determinada u otras validaciones. Con Spring y el API de JavaEE en el paquete `javax.validation`, no sólo podemos reducir este código sino además centralizarlo y simplificar el proceso.

Asumamos que requerimos las siguientes validaciones para el contacto que queremos crear:

1. El nombre del contacto es requerido (no puede ser nulo)
2. El nombre del contacto debe tener por lo menos 2 caracteres de longitud y máximo 30 caracteres
3. El número de teléfono empieza con el símbolo “+” seguido de por lo menos un dígito y únicamente admite una secuencia de dígitos entre el 0 y el 9, sin espacios ni separadores

Estas restricciones se agregan mediante anotaciones a los campos de la clase que queremos verificar, en nuestro caso, `ContactRequest`, usando anotaciones del paquete `javax.validation.constraint`, el cual te recomendamos examinar para ver más validaciones útiles:

```
1 @NotNull(message="El nombre es requerido")
2 @Size(min=2, max=30, message="El nombre debe tener entre {min} y {max} caracteres")
3 private String firstName;
4 @Pattern(regexp="^[0-9]*$", message="El número de telefono sólo puede tener dígitos
5 iniciando con el símbolo +")
```

La validación automática la implementas agregando la anotación *@Valid* junto al parámetro que quieras validar, siempre y cuando la Clase respectiva tenga las anotaciones que

acabamos de ver. Esto lo podemos hacer en el RestController:

```
1 public ContactResponse updateOrSave(@RequestBody @Valid ContactRequest contactRequest)
```

Este es un fragmento de una respuesta a una petición no válida:

```
1  {...
2  "defaultMessage": "El nombre debe tener entre 2 y 30 caracteres",
3  "objectName": "contactRequest",
4  "field": "firstName",
5  "rejectedValue": "",
6  "bindingFailure": false,
7  "code": "Size"
8  ...}
```

## Conectando con una base de datos externas

Hasta este momento, todos los registros que guardamos en base de datos mediante las peticiones POST han sido guardados en una base de datos en memoria que se borra cuando se finaliza la aplicación. ¡Aunque esto es muy útil para las pruebas unitarias y durante la etapa de desarrollo... de poco sirve para una aplicación en producción, por eso vamos a ver como conectarse a una base de datos externa... manteniendo también la opción en memoria! Lo mejor de ambos mundos.

## Perfiles y Ambientes

Una de mis funcionalidades favoritas de Spring es la posibilidad de mantener configuraciones diferentes, por ejemplo, para usar en ambientes diferentes y que se pueden seleccionar dinámicamente al iniciar la aplicación. De esta manera podemos usar la base de datos en memoria (H2) durante el desarrollo y una base de datos externa como MySQL durante las pruebas de integración, las pruebas de aceptación u otro ambiente como *staging* o producción.

La configuración que cambia entre ambientes se debe mantener en un archivo de propiedades para cada ambiente. Cada uno de estos archivos contiene una lista de propiedades que se conoce como perfil y tiene un nombre o selector asociado. Estas propiedades reemplazan a la configuración por defecto si se especifican.

Por defecto, Spring Boot se configura con una base de datos H2 en memoria y suponiendo que nos interesa cambiar esta configuración sólo en un ambiente de UAT (Pruebas de Aceptación) para conectarse a una base de datos MySQL, entonces debemos crear un archivo de propiedades con un selector “UAT” y las propiedades de conexión a base de datos que reemplacen la configuración por defecto.

Para hacerlo, ve al directorio “*src/main/resources*” y crea un archivo de nombre “*application-uat.properties*”. El selector o nombre de perfil es el texto que está después del guión (“-”) y antes del punto, en este caso “uat”. Puede ser cualquier texto siempre que sea un nombre de archivo válido. En ese archivo, escribe lo siguiente:

```
1spring.datasource.platform=mysql
2spring.datasource.url=jdbc:mysql://localhost:3306/contacts_db
3spring.datasource.username=contacts_user
4spring.datasource.password=contacts_password
```

En este ejemplo asumimos que tienes una base de datos MySQL de nombre “contacts\_db”, instalada en la misma máquina en la que se ejecuta tu microservicio y que puedes acceder

con el usuario “contacts\_user” y password “contacts\_password”. Las propiedades se explican solas con excepción de *spring.datasource.platform* que sirve para indicarle a Spring cual base de datos usar en caso de que encuentre varios drivers JDBC en el classpath. Hasta el momento sólo tenemos el driver H2 que incluimos al crear el proyecto en la primera parte de este extenso post. Incluye la dependencia Maven para el driver de MySQL y ahora los tendrás ambos. Mira el pom.xml nuestro repositorio en GitHub.

Solo falta indicarle a Spring que por defecto utilice la base de datos H2, pues ahora debemos elegir entre H2 y MySQL. Para ello, en el archivo “application.properties” agrega la siguiente línea:

```
spring.datasource.platform=h2
```

Para seleccionar el perfil activo hay varias maneras que incluyen parámetros de la línea de comandos y variables de entorno. Usaremos ésta última: crea una variable de entorno del sistema operativo que se llame “*SPRING\_PROFILES\_ACTIVE*” con valor “uat” e inicia la aplicación. Si todo va bien deberías ver una línea como la siguiente en el log de inicio de la aplicación, indicando el perfil activo (uat):

```
INFO 20841 --- [ main] c.s.contacts.ContactsWsApplication : The following profiles are active: uat
```

## Conclusión

Con los dos primeros laboratorios ya tendrías suficiente para tener un microservicio funcionando, pero el propósito de estos laboratorios es ayudarte a ir más allá y conseguir aplicaciones a prueba de balas. En estos laboratorios hemos cubierto varios aspectos básicos, intermedios y avanzados que no son exclusivos de Spring Boot pero que en cualquier caso te ayudarán a construir un microservicio u otro tipo de aplicación basada en Spring, que sea modificable, fácil de mantener, fácil de configurar y que esté preparada para ser usada en pruebas automatizadas.