

Arquitectura e implementación de Microservicios con Spring Cloud Netflix OSS

ISC. Ivan Venor García Baños
Instructor





Agenda

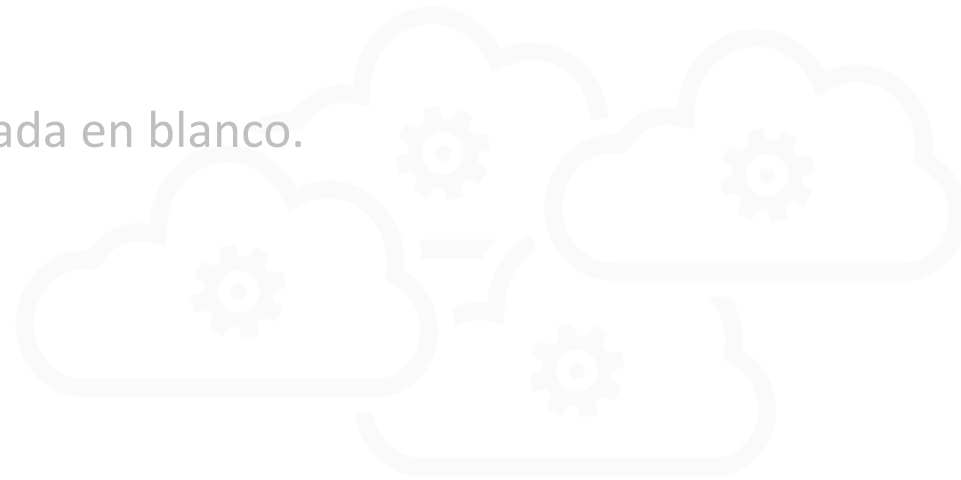
1. Presentación
2. Objetivos
3. Contenido
4. Despedida



Microservices



Esta página fue intencionalmente dejada en blanco.



Microservices



3. Contenido

- i. Arquitectura de sistemas monolíticos
- ii. Introducción a la Arquitectura Orientada a Servicios
- iii. Fundamentos Spring Boot 2.x
- iv. Arquitectura de Microservicios
- v. Microservicios con Spring Cloud y Spring Cloud Netflix OSS

Microservices



3. Contenido

- i. Arquitectura de sistemas monolíticos
- ii. Introducción a la Arquitectura Orientada a Servicios
- iii. **Fundamentos Spring Boot 2.x**
- iv. Arquitectura de Microservicios
- v. Microservicios con Spring Cloud y Spring Cloud Netflix OSS

Microservices



iii. Fundamentos Spring Boot 2.x



Microservices



iii. Fundamentos Spring Boot 2.x

iii.i Introducción a Spring Boot

iii.ii Configuración de propiedades en Spring Boot

iii.iii Perfiles

iii.iv Spring MVC

iii.v Spring Data JPA

iii.vi Spring Data REST

iii.vii Spring Boot Actuator

Microservices



iii.i Introducción a Spring Boot



Microservices



Objetivos de la lección

iii.i Introducción a Spring Boot

- Aprender como iniciar un proyecto desde cero con Spring Boot.
- Conocer las principales herramientas para el rápido “up-and-running” de una aplicación con Spring Boot.
- Conocer que son los “Bill-of-materials” (BOM) y los “Starters” que manejan las dependencias en Spring Boot.
- Iniciar un aplicativo simple con Spring Boot.

Microservices



iii.i Introducción a Spring Boot (a)

- Spring vs Spring Boot.
- Spring Framework es el Framework defacto para el desarrollo de aplicaciones Java empresariales, sin embargo requiere:
 - Amplia experiencia en el manejo de dependencias del proyecto de Spring e integración de proveedores externos como Hibernate, Quartz, Jackson, JavaMail, etc.
 - Configuración desde cero. Es necesario configurar todos los beans de infraestructura (boilerplate code).
 - Spring Framework es un conjunto de librerías, módulos, buenas prácticas, clases, etc, sin embargo no define como deben de ser configuradas dada su alta flexibilidad.



iii.i Introducción a Spring Boot (b)

- Spring Framework:
 - No exige un estandar de desarrollo y ni de configuración.
 - Es altamente flexible.
- Integrar otros frameworks y sub-proyectos de Spring cada vez se hace más complicado.





iii.i Introducción a Spring Boot (c)

- Spring Boot es un proyecto “umbrella” que esta construido sobre Spring Framework.
- Ha sido desarrollado por la misma comunidad de Spring Framework en el cuál colaboran los más exitosos desarrolladores Java open-source de la industria aplicando las mejores prácticas de desarrollo.
- Spring Boot es:
 - Opinonado debido a que está basado en un estándar de desarrollo dirigido por sus colaboradores.
 - Hacer “pair-programming” con el equipo de Spring Boot.



iii.i Introducción a Spring Boot (d)

- Spring Boot:



Josh Long (龙之春, जोश) ✓
@starbuxman

Following

a reminder: @SpringBoot lets you pair-program with the #Spring team.

7:28 PM - 21 Sep 2017 from San Antonio, TX

Microservices



iii.i Introducción a Spring Boot (e)

- El principal objetivo de Spring Boot es:
 - Permitirle a los desarrolladores crear aplicaciones productivas basadas en Spring Framework con el mínimo esfuerzo requerido, para su configuración, basandose en la experiencia de los colaboradores del proyecto Spring Boot.

Microservices



iii.i Introducción a Spring Boot (f)

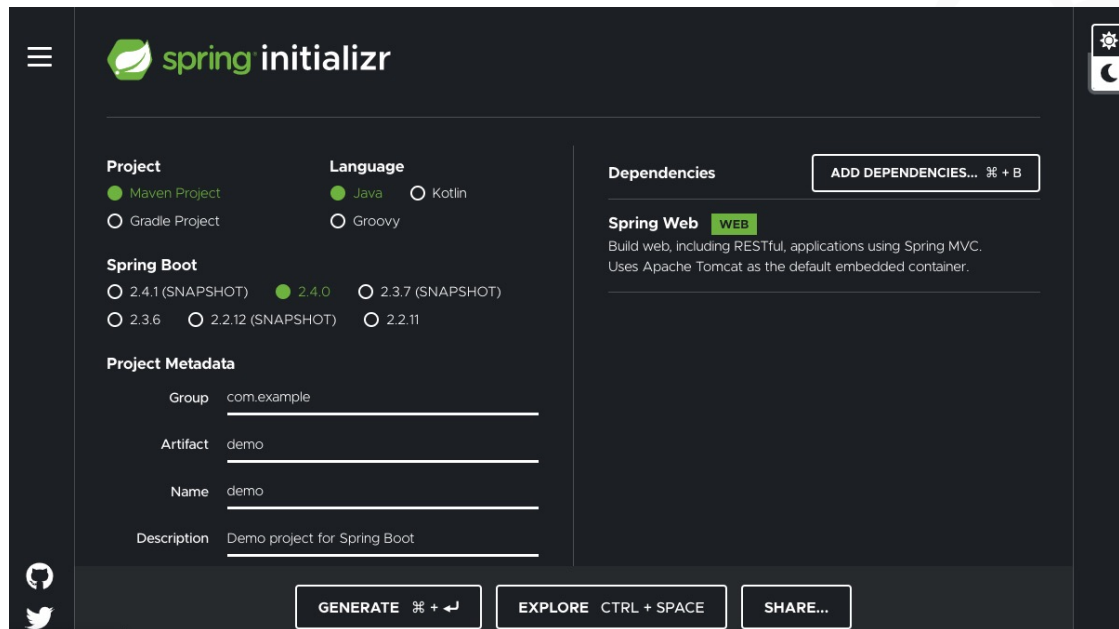
- Spring Boot.





iii.i Introducción a Spring Boot (g)

- Spring Initializr: Es una interface web para inicializar un proyecto con Spring Boot:



The screenshot shows the Spring Initializr web interface. It features a dark theme with a sidebar on the left containing a menu icon and social media links. The main content area is divided into several sections:

- Project:** Radio buttons for ☒ Maven Project and ☐ Gradle Project.
- Language:** Radio buttons for ☒ Java and ☐ Kotlin, with ☐ Groovy below.
- Spring Boot:** Radio buttons for versions: ☐ 2.4.1 (SNAPSHOT), ☒ 2.4.0, ☐ 2.3.7 (SNAPSHOT), ☐ 2.3.6, ☐ 2.2.12 (SNAPSHOT), and ☐ 2.2.11.
- Project Metadata:** Text input fields for Group (com.example), Artifact (demo), Name (demo), and Description (Demo project for Spring Boot).
- Dependencies:** A section with a button "ADD DEPENDENCIES... ⌘ + B". Below it, "Spring Web" is selected with a green "WEB" tag. A description reads: "Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container."

At the bottom, there are three buttons: "GENERATE ⌘ + ↵", "EXPLORE CTRL + SPACE", and "SHARE...".

<https://start.spring.io/>



iii.i Introducción a Spring Boot (h)

- **Práctica 1. Spring Initializr**
- Inicializar un proyecto Spring Boot mediante Spring Initializr.
- Generar un proyecto con las siguientes características:
 - Maven Project con Java y usando Spring Boot 2.1.3 (o la más actual, NO SNAPSHOT)
 - Group: **com.truper.springboot**
 - Artifact: **1-Spring-Initializr**
 - Dependencias: **Web**
 - Package Name: **com.truper.springboot.practica1**
 - Generar proyecto e importar en eclipse / STS.
- Implementar un **@RestController** simple, ejecutar el proyecto y abrir navegador <http://localhost:8080>



iii.i Introducción a Spring Boot (i)

- Spring Boot CLI.
- Herramienta de línea de comandos para crear un prototipo de proyecto basado en Spring Boot “production-ready”; es similar a Spring Initializr.
- Requiere instalación:
 - <https://docs.spring.io/spring-boot/docs/current/reference/html/getting-started-installing-spring-boot.html#getting-started-installing-the-cli>

Microservices



iii.i Introducción a Spring Boot (j)

- Spring Boot con Spring Boot CLI



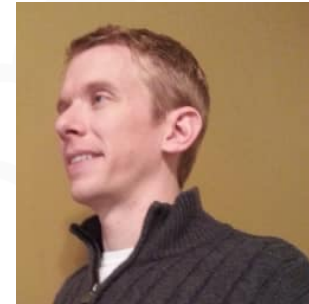
Rob Winch
@rob_winch

@Controller

```
class ThisWillActuallyRun {
    @RequestMapping("/")
    @ResponseBody
    String home() {
        "Hello World!"
    }
}
```

15:12 - 6 ago. 2013

https://twitter.com/rob_winch/status/364871658483351552?lang=es





iii.i Introducción a Spring Boot (k)

- BOM – Bill of Materials (Lista de materiales)
 - **¿Qué es un POM? (Maven)**
 - Project Object Model.
 - Archivo XML que contiene información y configuración del proyecto, utilizado por Maven, para importar sus dependencias.
 - **¿Qué es un BOM?**
 - Es un tipo especial de POM que es utilizado para controlar las versiones de las dependencias de un proyecto de forma centralizada.
 - Provee flexibilidad para agregar una dependencia evitando la preocupación referente a la versión compatible de la misma con el resto de dependencias.



iii.i Introducción a Spring Boot (I)

- BOM – Bill of Materials (Lista de materiales)
 - **¿Cómo utilizar un BOM en nuestro proyecto?**
 - Hacer que nuestro proyecto herede de un POM padre (BOM).
 - Un proyecto puede heredar de un solo proyecto padre, no es eficiente en definiciones de proyectos amplios y complejos (por ejemplo: Requiere Spring Cloud).
 - Importar uno o varios BOMs tal como sea requerido.

Microservices



iii.i Introducción a Spring Boot (m)

- Heredar de BOM padre (a):

```
<project ... >
```

```
...
```

```
<parent>
```

```
<groupId>org.springframework</groupId>
```

```
<artifactId>spring-framework-bom</artifactId>
```

```
<version>5.1.5.RELEASE</version>
```

```
<type>pom</type>
```

```
<scope>import</scope>
```

```
</parent>
```

```
...
```

Heredar de POM (BOM)
padre

Microservices



iii.i Introducción a Spring Boot (n)

- Heredar de BOM padre (b):

```
...  
<dependencies>  
    <dependency>  
        <groupId>org.springframework</groupId>  
        <artifactId>spring-context</artifactId>  
    </dependency>  
    <dependency>...</dependency>  
</dependencies>  
</project>
```

No requiere especificar versión.

Microservices



iii.i Introducción a Spring Boot (ñ)

- Importación de BOM (a):

```
<project ... >
```

```
...
```

```
<dependencies>
```

```
  <dependency>
```

```
    <groupId>org.springframework</groupId>
```

```
    <artifactId>spring-context</artifactId>
```

```
  </dependency>
```

```
  <dependency>...</dependency>
```

```
</dependencies>
```

```
...
```

No requiere
especificar versión.



iii.i Introducción a Spring Boot (o)

- Importación de BOM (b):

```

...
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-framework-bom</artifactId>
      <version>5.1.5.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
</project>

```

Importación de BOM



iii.i Introducción a Spring Boot (p)

- Starters.
- Son un conjunto de descriptores de dependencias que se incluirán en el proyecto si dicho “starter” es incluido en el mismo.
- Los “starters” evitan que el desarrollador tenga que preocuparse por incluir todas las dependencias relacionadas a una funcionalidad, implementada con Spring en particular.
- Si el proyecto requiere Spring Data JPA para implementar acceso a datos, únicamente es necesario incluir la dependencia “spring-boot-starter-data-jpa” al proyecto.



iii.i Introducción a Spring Boot (q)

- Existen múltiples “starters” proveídos por la comunidad de Spring Boot listos para ser incluidos en un proyecto Java empresarial con Spring Boot:
 - <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-starter>
- spring-boot-starter
- spring-boot-starter-activemq
- spring-boot-starter-amqp
- spring-boot-starter-aop
- spring-boot-starter-data-elasticsearch
- spring-boot-starter-data-jdbc
- spring-boot-starter-data-jpa
- spring-boot-starter-data-mongodb
- spring-boot-starter-data-redis
- spring-boot-starter-web
- spring-boot-starter-webflux
- spring-boot-starter-data-rest
- spring-boot-starter-oauth2-client



iii.i Introducción a Spring Boot (r)

- Creando un jar ejecutable.
- Spring Boot genera aplicaciones autocontenidas que incluyen todo lo necesario para poder ejecutarse, también llamadas aplicaciones “fat jar” o “uber jar” las cuales, por default, son empaquetadas como “jar”.
- Existen 3 modos de generar aplicaciones Spring Boot autocontenidas:
 - Spring CLI mediante comando “**spring jar <nombre-de-jar> <archivos>**”
 - Maven mediante **spring-boot-maven-plugin** starter y comando: “**mvn package**”
 - Gradle mediante “**spring-boot**” plugin y comando “**gradle build**” (fuera de alcance).



iii.i Introducción a Spring Boot (s)

- Ventajas de crear una aplicación autocontenida:
 - No requiere más dependencias para ejecutarse.
 - Se ejecuta mediante “java -jar”.
 - Se ejecuta donde sea que contenga una JVM.
 - No requiere servidor de aplicaciones.
 - Facilita el diseño de arquitecturas distribuidas.
 - Ideal para arquitecturas de Microservicios.
 - Ideal para empaquetar y desplegar en la nube. (PaaS y/o IaaS).



Microservices



iii.i Introducción a Spring Boot (t)

- **Práctica 2. Creando un jar ejecutable**
- Inicializar un proyecto Spring Boot mediante Spring CLI.
- Instala Spring CLI.
- Crea el fichero: **{tu-workspace}/2-Spring-CLI/AlumnosController.java**
- Implementar el controlador AlumnosController, ejecuta el comando:
 “spring run AlumnosController.java” y,
 abrir navegador: <http://localhost:8080>
- Ingresar a la ruta: **{tu-workspace}/2-Spring-CLI**
- Ejecuta los comandos:
 “spring jar miApp.jar AlumnosController.java” después,
 “java -jar miApp.jar” y,
 abrir navegador: <http://localhost:8080>



iii.i Introducción a Spring Boot (u)

- Configuración Spring Boot.
- Spring Boot habilita auto-configuración mediante el escaneo automático de dependencias que han sido agregadas al proyecto, es decir, genera los beans necesarios para el proyecto en base a las dependencias existentes en el classpath.
 - Ejemplo: Si la librería (dependencia) H2 se encuentra en el classpath del proyecto y no se ha definido ninguna configuración de beans relacionada a un DataSource, Spring Boot **auto-configurará** una base de datos en memoria utilizando el motor H2.



iii.i Introducción a Spring Boot (v)

- Configuración Spring Boot.
- La auto-configuración que aplica Spring Boot es no es invasiva, es decir, es posible definir una configuración propia del proyecto simplemente definiendo los beans correspondientes:
 - Ejemplo: Si se define un bean DataSource, la **auto-configuración** de Spring Boot para definir un DataSource en memoria (embebida) no surtirá efecto.

Microservices



iii.i Introducción a Spring Boot (w)

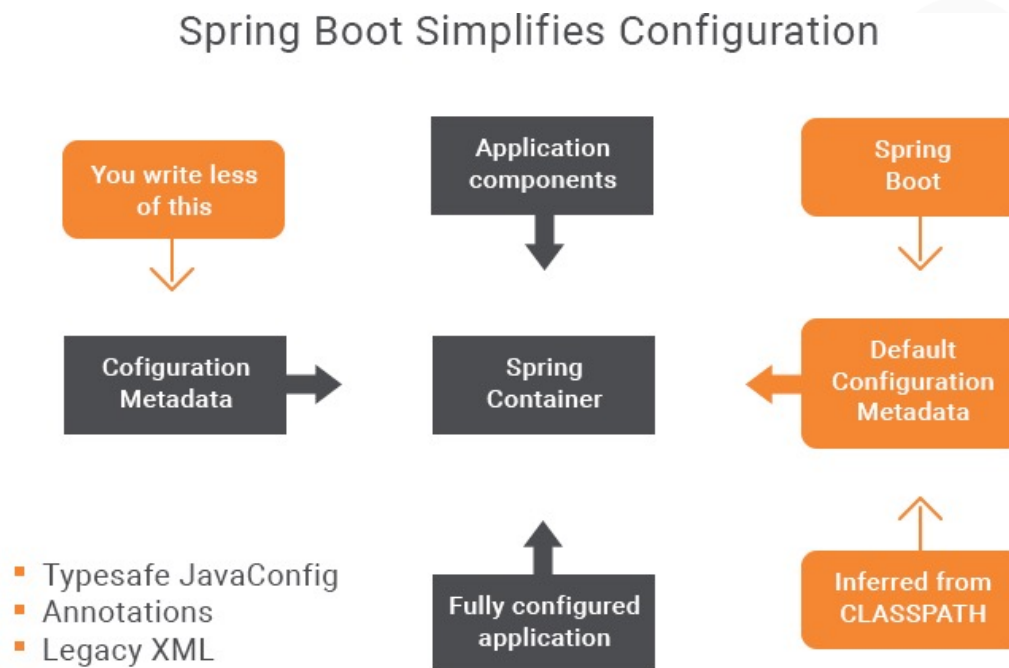
- Configuración Spring Boot.
- Para habilitar la **auto-configuración**, de Spring Boot, únicamente es necesario agregar las anotaciones **@EnableAutoConfiguration** o **@SpringBootApplication** a la clase de configuración primaria del proyecto (definir @EnableAutoConfiguration o @SpringBootApplication una sola vez).

Microservices



iii.i Introducción a Spring Boot (x)

- Configuración Spring Boot.





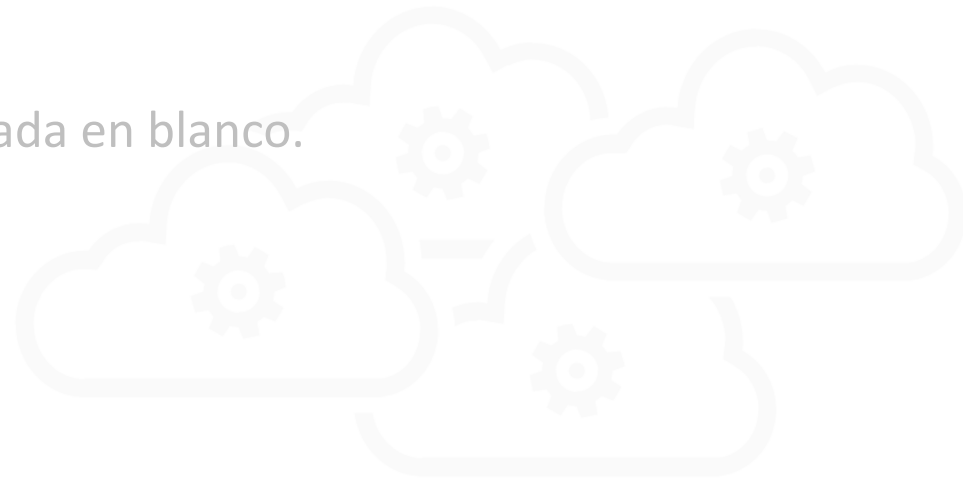
Resumen de la lección

iii.i Introducción a Spring Boot

- Comprendimos las diferencias entre Spring y Spring Boot.
- Aprendimos a utilizar las herramientas Spring Initializr y Spring CLI para crear proyectos Spring Boot desde cero.
- Comprendimos que el objetivo de Spring Boot es inicializar un aplicativo Java Empresarial sin la complejidad de su configuración.
- Revisamos que es el BOM y los Starters.
- Creamos un aplicativo Spring Boot autocontenido en un jar, mediante la aplicación de las herramientas Spring Initializr y Spring CLI.



Esta página fue intencionalmente dejada en blanco.



Microservices



iii. Fundamentos Spring Boot 2.x

iii.i Introducción a Spring Boot

iii.ii Configuración de propiedades en Spring Boot

iii.iii Perfiles

iii.iv Spring MVC

iii.v Spring Data JPA

iii.vi Spring Data REST

iii.vii Spring Boot Actuator



Microservices



iii.ii Configuración de propiedades en Spring Boot

Microservices



Objetivos de la lección

iii.ii Configuración de propiedades en Spring Boot

- Comprender como se configura a nivel propiedades, una aplicación Spring Boot.
- Analizar los diferentes mecanismos de configuración mediante archivo de propiedades y YAML.
- Exponer configuración externalizada.

Microservices



iii.ii Configuración de propiedades en Spring Boot (a)

- application.properties y configuración YAML.
- Spring Boot carga sus propiedades de múltiples formas sin embargo, el archivo “**application.properties**” es el archivo de propiedades por default de Spring Boot.
- Spring Boot carga las propiedades, del archivo “**application.properties**” y las agrega al objeto **Environment** de Spring.

Microservices



iii.ii Configuración de propiedades en Spring Boot (b)

- application.properties y configuración YAML.
- La ubicación del archivo “**application.properties**” se define por la siguiente precedencia:
 - Subdirectorio /config del directorio actual.
 - El directorio actual
 - En el paquete /config del classpath
 - En el directorio raíz del classpath
- Las propiedades definidas en ubicaciones superiores sobre-escriben aquellas definidas en ubicaciones inferiores.



iii.ii Configuración de propiedades en Spring Boot (c)

- Acceso a propiedades por línea de comando.
- Por default Spring Boot convierte cualquier argumento de línea de comandos (iniciando con --) como una propiedad y la agrega al objeto Environment: Ejemplo: **--server.port=9090**
- Es posible cambiar el nombre del archivo “**application.properties**” utilizando la propiedad **spring.config.name** pasada como parámetro al ejecutar la aplicación Spring Boot.
 - **java -jar myproject.jar --spring.config.name=myproject**
 - El comando anterior buscará un archivo de propiedades llamado “**myproject.properties**” en el orden de precedencia por default.



iii.ii Configuración de propiedades en Spring Boot (d)

- application.properties
- Spring Boot define un conjunto de propiedades básicas para habilitar auto-configuración:
 - # Spring Config
 - spring.config.location
 - spring.config.name
 - # Email
 - spring.mail.default-encoding
 - spring.mail.host
 - spring.mail.password
 - # Profiles
 - spring.profiles.active
 - spring.profiles.include
 - # Embedded Server Configuration
 - server.port
 - server.servlet.context-path

<https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html>



iii.ii Configuración de propiedades en Spring Boot (e)

- Configuración YAML
- Es posible utilizar el formato YAML en lugar de utilizar “**properties**” siempre y cuando la dependencia SnakeYAML se encuentre en el classpath.
 - **spring-boot-starter** agrega la dependencia SnakeYAML.
- YAML es un subconjunto de JSON y es preferido al especificar configuración de propiedades de forma jerárquica (opinionado).
- YAML es más “humanamente leíble” sin embargo, es opinionado.



iii.ii Configuración de propiedades en Spring Boot (f)

- Configuración YAML
- Ejemplo de application.yml:
myapp:
 name: Ivan
 age: 31
- Como archivo application.properties:
myapp.name=Ivan
myapp.age=31
myapp.name.last=Garcia
- Propiedades definidas por archivos “**properties**” tiene mayor precedencia a las definidas por archivos YAML.

Microservices



iii.ii Configuración de propiedades en Spring Boot (g)

- Spring Boot permite externalizar la configuración del aplicativo de modo que sea posible ejecutar el mismo código en distintos ambientes:
 - Configuración de conexión a Base de Datos por diferentes ambientes.
 - Configuración de servidor SFTP por diferentes ambientes, etc.
- La configuración puede definirse de diferentes modos:
 - Archivos de propiedades.
 - Archivos YAML.
 - Variables de entorno.
 - Argumentos de línea de comandos.



iii.ii Configuración de propiedades en Spring Boot (h)

- Los valores de las propiedades pueden ser inyectadas directamente sobre las propiedades de los Beans mediante la anotación **@Value("\${nombre-de-la-propiedad}")** o accediendo a ellas a través del objeto **Environment** de Spring (**env.getProperty("nombre-de-la-propiedad")**).

Microservices



iii.ii Configuración de propiedades en Spring Boot (i)

- Precedencia en el que se busca la configuración externalizada (a):
 1. Configuración global Devtools: En el directorio raíz del sistema (~/.spring-boot-devtools.properties), sólo si DevTools esta activo.
 2. Anotación **@TestPropertySource** sobre clases de Test.
 3. Atributos "**properties**" sobre tests, disponibles mediante **@SpringBootTest**.
 - ✓ 4. Argumentos de línea de comandos (--argument=value).
 - ✓ 5. Propiedades desde variable de entorno **SPRING_APPLICATION_JSON** (JSON embebido en línea como variable de entorno o propiedad del sistema).
 6. Parámetros de inicialización de configuración de Servlet (ServletConfig init parameters).
 7. Parámetros de inicialización de contexto de Servlet (ServletContext init parameters).



iii.ii Configuración de propiedades en Spring Boot (j)

- Precedencia en el que se busca la configuración externalizada (b):

8. Atributos JNDI desde: **java:comp/env**.
9. Propiedades Java del sistema: **System.getProperties()**.
- ✓ 10. Variables de entorno del Sistema Operativo (OS environment variables).
11. Propiedades Random (**random.***).
12. application.properties específicas de Perfil desde fuera del empaquetado del aplicativo/jar (**application-{profile}.properties incluye variantes YAML**).
13. application.properties específicas de Perfil desde dentro del empaquetado del aplicativo/jar (**application-{profile}.properties incluye variantes YAML**).



iii.ii Configuración de propiedades en Spring Boot (k)

- Precedencia en el que se busca la configuración externalizada (c):

- ✓ 14. application.properties desde fuera del empaquetado del aplicativo/jar (**application.properties incluye variantes YAML**).
- ✓ 15. application.properties desde dentro del empaquetado del aplicativo/jar (**application.properties incluye variantes YAML**).
- 16. Anotaciones **@PropertySource** sobre clases de configuración **@Configuration**.
- 17. Propiedades default especificadas por **SpringApplication.setDefaultProperties**.

- Nota: No se revisarán todas las formas de precedencia de configuración externalizada, sólo las más comunes.

<https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-external-config.html#boot-features-external-config>



iii.ii Configuración de propiedades en Spring Boot (I)

- **Práctica 3. Configuración Externalizada**
- Instala lombok en tu IDE.
- Ingresar a la ruta: **{tu-workspace}/3-Spring-Configuracion-Externalizada**
- Ejecutar el proyecto ejecutando la clase principal y abrir navegador en:
<http://localhost:8080>
- Implementar diferentes métodos de configuración externalizada de propiedades:
 - controller.message
 - server.port



iii.ii Configuración de propiedades en Spring Boot (m)

- Práctica 3. Configuración Externalizada

1. Analizar la clase principal Application.java, ejecutar el proyecto.
2. Probar <http://localhost:8080>
3. Crear archivo “**application.yml**” y definir propiedades **server.port: 8081** y **controller.message: Mi mensaje desde application.yml**. Ejecutar el proyecto.
4. Probar <http://localhost:8081> (nótese que el puerto 8080 ya no responde).
5. Definir propiedades, en archivo “**application.properties**”, **server.port=8085** y **controller.message=Mi mensaje desde application.properties**. Ejecutar el proyecto.
6. Probar <http://localhost:8081> y <http://localhost:8085>



iii.ii Configuración de propiedades en Spring Boot (n)

- Práctica 3. Configuración Externalizada

7. Definir la variable de sistema (variable de entorno) **SERVER_PORT=9090** y **CONTROLLER_MESSAGE='Mi mensaje desde variable del sistema'** dependiendo del sistema operativo host.
 - Ejemplo, en Mac:
 - ✓ Editar ~/.bashrc, agregar variables del sistema:
 - ✓ export SERVER_PORT=9090
 - ✓ export CONTROLLER_MESSAGE='Mi mensaje desde variable del sistema'
 - ✓ Recargar el archivo ~/.bash_profile (source ~/.bash_profile)
8. Compilar proyecto mediante **mvn clean package** y ejecutar mediante comando:
java -jar target/3-Spring-Configuracion-Externalizada-0.0.1-SNAPSHOT.jar
9. Probar <http://localhost:9090>



iii.ii Configuración de propiedades en Spring Boot (ñ)

- Práctica 3. Configuración Externalizada

7. Definir la variable de sistema (variable de entorno) **SERVER_PORT=9090** y **CONTROLLER_MESSAGE='Mi mensaje desde variable del sistema'** dependiendo del sistema operativo host.
 - Ejemplo, en Windows:
 - ✓ Editar variables del sistema:
 - ✓ Variable: SERVER_PORT Valor: 9090
 - ✓ Variable: CONTROLLER_MESSAGE Valor: Mi mensaje desde variable del sistema.
8. Compilar proyecto mediante **mvn clean package** y ejecutar mediante comando:
java -jar target/3-Spring-Configuracion-Externalizada-0.0.1-SNAPSHOT.jar
9. Probar <http://localhost:9090>



iii.ii Configuración de propiedades en Spring Boot (o)

- Práctica 3. Configuración Externalizada

10. Definir la variable de sistema (variable de entorno)

**SPRING_APPLICATION_JSON='{"controller":{"message":"Mi mensaje desde
SPRING_APPLICATION_JSON"},"server":{"port":9099}}'** dependiendo del
sistema operativo host.

11. Compilar proyecto mediante **mvn clean package** y ejecutar mediante comando:

java -jar target/3-Spring-Configuracion-Externalizada-0.0.1-SNAPSHOT.jar

12. Probar <http://localhost:9099>

Microservices



iii.ii Configuración de propiedades en Spring Boot (p)

- Práctica 3. Configuración Externalizada

13. Por último, para comprobar la precedencia de la configuración externalizada, compilar proyecto mediante **mvn clean package**.
14. Ejecutar el proyecto mediante comando:
java -jar target/3-Spring-Configuracion-Externalizada-0.0.1-SNAPSHOT.jar
--controller.message="Mi mensaje desde línea de comando."
--server.port=9001
15. Probar <http://localhost:9001>



Resumen de la lección

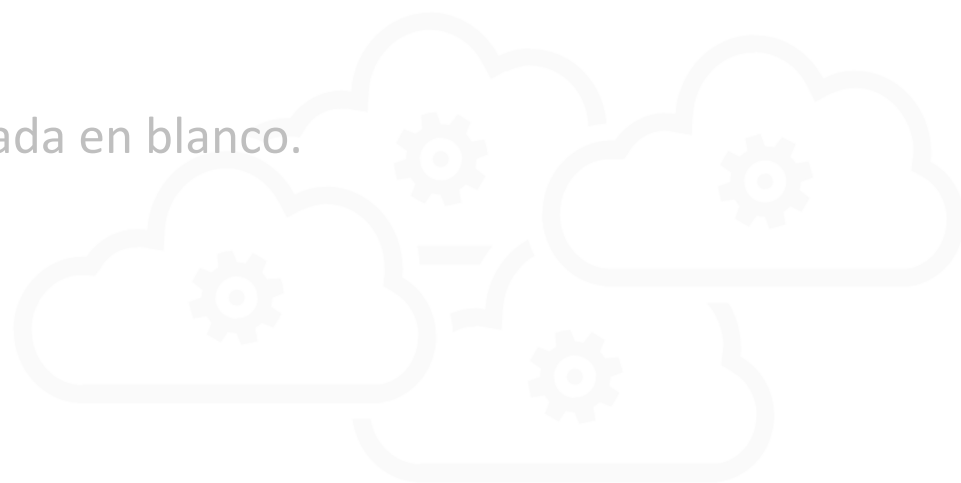
iii.ii Configuración de propiedades en Spring Boot

- Comprendimos las diferencias entre configuración mediante archivos “properties” y YAML.
- Revisamos la precedencia de los archivos de configuración para la implementación de configuración externalizada.
- Analizamos algunas propiedades de Spring Boot que facilitan la configuración de la aplicación.

Microservices



Esta página fue intencionalmente dejada en blanco.



Microservices



iii. Fundamentos Spring Boot 2.x

iii.i Introducción a Spring Boot

iii.ii Configuración de propiedades en Spring Boot

iii.iii **Perfiles**

iii.iv Spring MVC

iii.v Spring Data JPA

iii.vi Spring Data REST

iii.vii Spring Boot Actuator



Microservices



iii.iii Perfiles



Microservices



Objetivos de la lección

iii.iii Perfiles

- Comprender para que se utilizan los perfiles o “profiles” en Spring.
- Aprender como activar perfiles dependiendo el contexto de ejecución del aplicativo.

Microservices



iii.iii Perfiles (a)

- Los perfiles o “**profiles**” es un mecanismo que permite al contenedor de beans de Spring habilitar el registro de diferentes beans de un mismo tipo para diferentes ambientes de ejecución, es decir, los perfiles permiten agrupar beans de una misma naturaleza o, beans de un mismo ambiente o entorno de ejecución.
- Un perfil es un identificador que permite agrupar definiciones de beans para ser registrados en el contenedor sólo si dicho perfil se encuentra activo.

Microservices



iii.iii Perfiles (b)

- Ejemplo: Es posible que se defina un bean de tipo DataSource con una base de datos en memoria H2 para un perfil de desarrollo o **"development"** y, definir, otro bean de tipo DataSource el cual adquiera su referencia a través de JNDI para un perfil de producción o **"production"**.

Microservices



iii.iii Perfiles (c)

- Mediante la anotación **@Profile** es posible indicar para que perfil corresponde un bean componente.
- El contenedor de beans de Spring sólo levantará los beans registrados en el perfil default (sin perfil) y aquellos registrados en los perfiles activos.
- Se pueden definir expresiones complejas a evaluar por los perfiles a través de los operadores lógicos “!” (not), “&” (and) y “|” (or).

Microservices



iii.iii Perfiles (d)

- Es posible agrupar operadores lógicos únicamente utilizando paréntesis.
 - La expresión = “**production & us-east | eu-central**” no es válida.
 - La expresión = “**production & (us-east | eu-central)**” es válida.
- Es posible utilizar la anotación `@Profile` como una meta-anotación para crear anotaciones personalizadas y “type-safe”.

```
@Retention(RetentionPolicy.RUNTIME)
```

```
@Profile("production")
```

```
public @interface Production { }
```

```
@Production
```

```
@Profile("production")
```



iii.iii Perfiles (e)

- @Profile

@Configuration

@Profile("dev")

```
public class StandaloneDataConfig {
```

@Bean

```
public DataSource dataSource() {
```

```
    return new EmbeddedDatabaseBuilder().setType(EmbeddedDatabaseType.HSQL)  
        .addScript("classpath:com/bank/config/sql/schema.sql")  
        .addScript("classpath:com/bank/config/sql/test-data.sql") .build();
```

```
}
```

```
}
```



iii.iii Perfiles (f)

- @Profile

@Configuration

@Production

```
public class IndiDataConfig {
```

```
    @Bean(destroyMethod="close")
```

```
    public DataSource dataSource() throws Exception {
```

```
        Context ctx = new InitialContext();
```

```
        return (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");
```

```
    }
```

```
}
```

Microservices



iii.iii Perfiles (g)

- Se utiliza la propiedad **spring.profiles.active** para habilitar un perfil o perfiles separados por coma (separados por coma implica “or”).
- También es posible, y más recomendable, pasar el perfil por línea de comando (**--spring.profiles.active**) o mediante variable de entorno del sistema.
- Definir la misma propiedad en múltiples archivos de configuración, ya sea por medio de **application.properties** o pasando el valor por **línea de comandos**, sigue la misma precedencia de como se cargan las propiedades.



iii.iii Perfiles (h)

- Archivos de configuración específicos por Perfil.
- Spring Boot permite la configuración de propiedades a través de archivos “**properties**” o archivos YAML.
- Estos archivos de configuración pueden tener configuración particular para un perfil específico.
- La convención para definir archivos “**properties**” o YAML específicos por perfil es **application-{profile}.properties** o **application-{profile}.yml**, dichos archivos se buscarán con la misma precedencia referente a la configuración externalizada.



iii.iii Perfiles (i)

- Archivos de configuración YAML multi-perfil.
- Es posible especificar multiple configuración de perfiles en un único archivo YAML. Para ello se utiliza la propiedad **spring.profiles**.

YAML	
server: address: 192.168.1.100	Configuración default

spring: profiles: development	Configuración perfil "development"
server: address: 127.0.0.1	

spring: profiles: production & eu-central	Configuración perfil "production & eu-central"
server: address: 192.168.1.120	



iii.iii Perfiles (j)

- Archivos de configuración YAML multi-perfil.
- Al utilizar configuración específica por perfil mediante YAML, sólo utilizar uno de los dos métodos expuestos para definir propiedades multi-perfil.
- En la práctica, Spring Boot sugiere utilizar configuración multi-perfil en un solo archivo YAML, debido a que es “humanamente leíble”.
- En la experiencia, se sugiere utilizar configuración mediante múltiples archivos específicos por perfil mediante “**properties**” (**application-{profile}.properties**).



iii.iii Perfiles (k)

- **Práctica 4. Perfiles y configuración multi-perfil**
- Ingresar a la ruta: **{tu-workspace}/4-Spring-Configuracion-Multi-Perfil**
- Analizar la clase `ConnectionDataBase`. ¿Qué bean `DummyDataSource` se va a inyectar? ¿valor de `connectionURL` se va a inyectar?
- Define la propiedad: **myapp.connection.url** con los valores requeridos para los perfiles dev, qa, staging y production.
- Define la clase `ProfilesConfig` y describe los 4 beans requeridos, de tipo `DummyDataSource`, para los ambientes dev, qa, staging y production.
- Poner en práctica los distintos modos de activación de perfiles.



Resumen de la lección

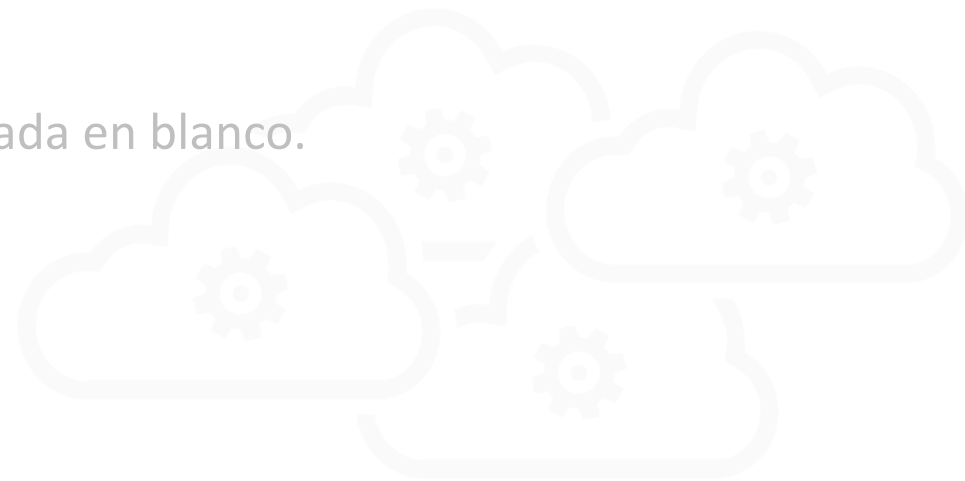
iii.iii Perfiles

- Comprendimos cómo implementar perfiles para el despliegue de aplicaciones Spring Boot en diferentes ambientes.
- Aprendimos como utilizar la anotación **@Profile** para definir beans de diferentes perfiles.
- Revisamos como activar perfiles.
- Verificamos como asociar perfiles a archivos de propiedades específicos mediante “**properties**” y YAML.

Microservices



Esta página fue intencionalmente dejada en blanco.



Microservices



iii. Fundamentos Spring Boot 2.x

iii.i Introducción a Spring Boot

iii.ii Configuración de propiedades en Spring Boot

iii.iii Perfiles

iii.iv Spring MVC

iii.v Spring Data JPA

iii.vi Spring Data REST

iii.vii Spring Boot Actuator

Microservices



iii.iv Spring MVC



Microservices



Objetivos de la lección

iii.iv Spring MVC

- Conocer la auto-configuración de Spring MVC mediante Spring Boot.
- Revisar a grandes rasgos las características que habilita Spring Boot a las aplicaciones Spring MVC.
- Conocer la convención sobre configuración principal de aplicaciones Spring MVC auto-configuradas con Spring Boot.
- Comprender como se implementa la negociación de contenido en aplicaciones Spring MVC con Spring Boot.
- Comprender cómo se integra Spring HATEOAS y cómo se habilita CORS a aplicaciones Spring MVC con Spring Boot.



iii.iv Spring MVC (a)

- Spring Boot habilita el desarrollo rápido de aplicaciones web mediante el módulo **"starter" spring-boot-starter-web**.
- Spring Boot permite crear aplicaciones web auto-contenidas con un servidor embebido siendo Tomcat, Jetty, Netty o Undertow, permitiendo empaquetar la aplicación en un jar y ejecutarlo en cualquier JVM, no necesita servidor web o servidor de aplicaciones.
- No es necesario configurar algún bean en particular para iniciar el desarrollo de una aplicación web, Spring Boot auto-configura los beans necesarios por default.



iii.iv Spring MVC (b)

- La auto-configuración de Spring Boot mediante el módulo “**starter**” **spring-boot-starter-web** incluye:
 - Inclusión de los beans **ContentNegotiatingViewResolver** y **BeanNameViewResolver**.
 - Soporte para servir contenido estático, así como recursos estáticos desde WebJars.
 - Registro automático de beans **Converter**, **GenericConverter** y **Formatter**.
 - Soporte de **HttpMessageConverters**.
 - Registro automático de **MessageCodesResolver**.
 - Soporte estático de templates.
 - Soporte de Favicon personalizado.
 - Configuración automática mediante bean **ConfigurableWebBindingInitializer**.



iii.iv Spring MVC (c)

- Mediante la inclusión del módulo “**starter**” **spring-boot-starter-web** es posible utilizar las anotaciones comunes al framework Spring Web MVC de forma tradicional y desarrollar una aplicación web.
 - @Controller
 - @RestController
 - @RequestMapping
 - @GetMapping
 - @PostMapping
 - @PutMapping
 - @DeleteMapping
 - @PatchMapping
 - @ResponseStatus
 - @ResponseBody
 - @RequestBody
 - @PathVariable
 - @RequestParam
 - @RequestHeader
 - @CookieValue
 - @RequestPart
 - @ModelAttribute
 - @SessionAttributes
 - @ControllerAdvice
 - @ExceptionHandler



iii.iv Spring MVC (d)

- Por default Spring Boot sirve contenido estático desde las ubicaciones **/static**, **/public**, **/resource**, **/META-INF/resources** o **/templates** del classpath o de la raíz del ServletContext.
- No se recomienda el uso de la ubicación standard **/src/main/webapp** cuando se trabaja con Spring Boot empaquetado como jar. La ubicación **/src/main/webapp** únicamente es considerada cuando la aplicación es empaquetada como war.
- Spring Boot por default muestra el archivo estático **index.html** como “welcome page”, el cual busca desde las ubicaciones mencionadas.



iii.iv Spring MVC (e)

- Content negotiation
- Spring Boot, como buena práctica, deshabilita la negociación del contenido de URL mediante sufijos y sugiere la utilización correcta del Header “Accept”.
 - La petición **GET /myproject/person.json** no se mapeará al handler mapping **@GetMapping(“/myproject/person”)**
- Por default Spring MVC, mapeaba la negociación del contenido mediante sufijo (file extension) en automático, dada la imposibilidad de los navegadores de manipular correctamente el Header “Accept”.



iii.iv Spring MVC (f)

- Content negotiation
- De no ser posible manipular correctamente el Header “Accept” por parte del cliente para negociar el contenido de respuesta, Spring Boot sugiere utilizar “query parameters” para la negociación del contenido.
 - La petición **GET /myproject/person?format=json** se mapeará al handler mapping **@GetMapping(“/myproject/person”)** si se habilita la propiedad:
spring.mvc.contentnegotiation.favor-parameter=true
 - Es posible cambiar el parámetro de negociación de contenido mediante la propiedad:
spring.mvc.contentnegotiation.parameter-name=formato



iii.iv Spring MVC (g)

- Templates
- Spring Boot habilita el uso y auto-configuración para la integración de templates para Spring MVC tales como: FreeMarker, Groovy, Thymeleaf y Mustache.
- JSP debe ser evitado debido a que se tiene prueba de limitaciones técnicas cuando se trabaja con contenedores web embebidos.
 - Con Jetty y Tomcat, JSP funciona con empaquetado war y ejecutando la aplicación mediante comando `java -jar <app>.war`.
 - Undertow no soporta JSPs.
 - La creación de un pagina de error customizada “error.jsp” no sobre-escribe la vista por default de manejo de errores (/error).



iii.iv Spring MVC (h)

- HATEOAS
- Spring Boot utiliza auto-configuración mediante la inclusión del módulo “**starter**” **spring-boot-starter-hateoas** sin mayor configuración, no necesita ser habilitado mediante `@EnableHypermediaSupport`.
- Habilita la auto-configuración del bean **ObjectMapper** de forma automática mediante propiedades **spring.jackson.***.
- Es posible crear la definición manual del bean **Jackson2ObjectMapperBuilder** para crear el bean **ObjectMapper**.



iii.iv Spring MVC (i)

- Soporte CORS
- Spring MVC permite el Cross-origin Resource Sharing o CORS mediante la anotación **@CrossOrigin**.
- La anotación **@CrossOrigin** puede utilizarse a nivel de método (handler methods) o a nivel de clase **@Controller** o **@RestController**.
- Spring Boot no requiere configuración adicional para habilitar CORS.
- **@CrossOrigin** habilita: todos los orígenes, todos los headers y todos los métodos HTTP que tenga mapeados el controlador.



+

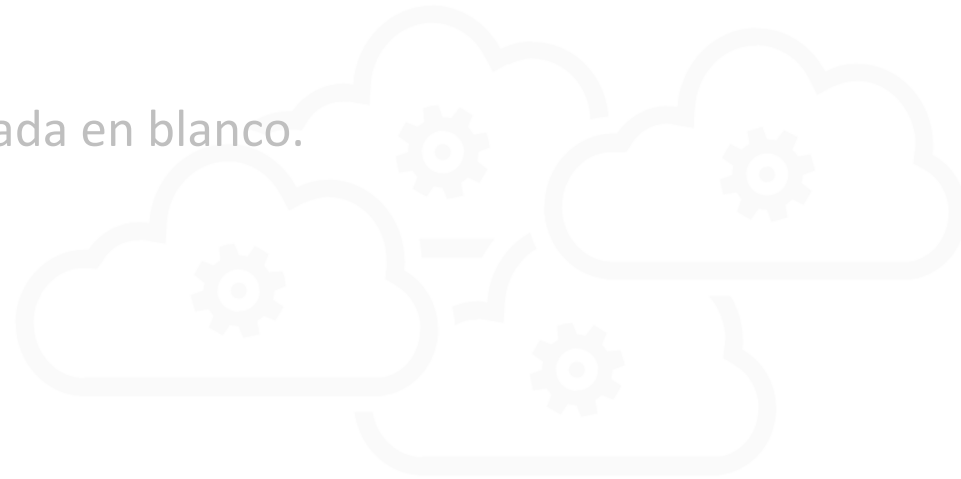
Resumen de la lección

iii.iv Spring MVC

- Comprendimos como habilitar auto-configuración de:
 - Spring MVC
 - Spring HATEOAS
- Comprendimos el mecanismo de negociación de contenido.
- Comprendimos cuales son las rutas pre-configuradas de Spring Boot para Spring MVC.
- Analizamos las debilidades de JSPs en aplicaciones Spring MVC auto-contenidas con Spring Boot.



Esta página fue intencionalmente dejada en blanco.



Microservices



iii. Fundamentos Spring Boot 2.x

- iii.i Introducción a Spring Boot
- iii.ii Configuración de propiedades en Spring Boot
- iii.iii Perfiles
- iii.iv Spring MVC
- iii.v Spring Data JPA**
- iii.vi Spring Data REST
- iii.vii Spring Boot Actuator

Microservices



iii.v Spring Data JPA



Microservices



Objetivos de la lección

iii.v Spring Data JPA

- Conocer la auto-configuración de Spring Data JPA mediante Spring Boot.
- Revisar a grandes rasgos las características que habilita Spring Boot a las aplicaciones Spring Data JPA.
- Conocer la convención sobre configuración principal de aplicaciones Spring Data JPA auto-configuradas con Spring Boot.
- Comprender como implementar Repositorios mediante Spring Data JPA.
- Comprender como implementar “**query methods**” en Spring Data JPA.



iii.v Spring Data JPA (a)

- Spring Framework provee extensivo soporte para trabajar con bases de datos SQL (y no SQL) mediante el acceso a través de JdbcTemplate (spring-jdbc) o integración completa mediante un ORM tal como Hibernate (spring-orm y hibernate).
- Spring Data es un proyecto construido sobre Spring Framework que provee un nivel más de abstracción sobre repositorios, el cual nos permite definir repositorios a partir de interfaces.
- Con Spring Data no es necesario implementar repositorios (o DAOs) una y otra vez conforme los requerimientos de las aplicaciones.



iii.v Spring Data JPA (b)

- Spring Data JPA trabaja sobre bases de datos relacionales o SQL, mediante el estándar JPA e implementación a través de Hibernate.
- Las aplicaciones Spring Data JPA auto-configuradas con Spring Boot 2.x utilizan Hibernate 5 como implementación JPA 2.0.
- Hibernate 3 y 4 han sido depreciadas en Spring Framework 5.0 y Spring Boot 2.x utiliza como código base Spring Framework 5.0.

Microservices



iii.v Spring Data JPA (c)

- Spring Boot habilita el desarrollo rápido de aplicaciones con Spring Data JPA mediante el módulo **"starter" spring-boot-starter-data-jpa** o **spring-boot-starter-jdbc** para el uso simple de Spring JDBC.
- Por default, Spring Boot habilita soporte para la creación de base de datos embebida (en memoria), si ningún bean de tipo **DataSource** ha sido configurado.
- Los motores de base de datos soportadas para implementar **"in-memory databases"** son: **HSQL, H2 y Derby**. Únicamente es necesario agregar la dependencia correspondiente, a la base de datos requerida, en el classpath.



iii.v Spring Data JPA (d)

- Spring Boot permite definir al desarrollador el DataSource requerido para el aplicativo y tomar todo el control referente al mismo.
- Definir nuestro propio bean **DataSource**, evitará la auto-configuración de Spring Data JPA mediante Spring Boot.

Microservices



iii.v Spring Data JPA (e)

- Conectividad a base de datos productiva.
- Spring Boot 2.x confía en la implementación de **DataSource** mediante “**pool**” de conexiones, lo cual habilita auto-configuración de DataSource productivos mediante los siguientes criterios:
 - Spring Boot seleccionará, en primer lugar, el driver **HikariCP** para crear pool de conexiones productivo, si la dependencia se encuentra en el classpath.
 - De no encontrarse **HikariCP**, utilizará el **JDBC Connection Pool** de **Tomcat**, si la dependencia se encuentra en el classpath.
 - Si no se encuentra **HikariCP** ni **Tomcat JDBC Connection Pool**, Spring Boot utilizará **Commons DBCP2**, para crear pool de conexiones, si se encuentra en el classpath.
 - No existen otras implementaciones de pool de conexiones que Spring Boot auto-configure.



iii.v Spring Data JPA (f)

- Conectividad a base de datos productiva.
- Si se utilizan los módulos “**starter**” **spring-boot-starter-jdbc** o **spring-boot-starter-data-jpa**, la dependencia **HikariCP** es agregada automáticamente.
- Es posible saltar el algoritmo de selección de **DataSource** por default especificando la propiedad **spring.datasource.type**, debido a que si la aplicación se ejecuta en un contenedor **Tomcat**, el tipo de DataSource (con pool de conexiones) será **tomcat-jdbc** por default.
- Es posible configurar otros **DataSource** manualmente.



iii.v Spring Data JPA (g)

- La auto-configuración del **DataSource** se da mediante configuración externalizada mediante las propiedades **spring.datasource.*** las cuales pueden definirse en el `application.properties`.
 - **spring.datasource.url=jdbc:mysql://localhost/test**
 - **spring.datasource.username=dbuser**
 - **spring.datasource.password=dbpass**
 - **spring.datasource.driverclass-name=com.mysql.jdbc.Driver**
- Al menos es necesario especificar la URL de conexión, de no definirse, Spring Boot tratará de auto-configurar una base de datos en memoria.
- Definir la propiedad **driverclass-name** es opcional.



iii.v Spring Data JPA (h)

- Spring Boot provee auto-configuración sobre **HikariCP**, **Tomcat JDBC Connection Pool** y **Commons DBCP2** mediante las propiedades **spring.datasource.hikari.***, **spring.datasource.tomcat.*** y **spring.datasource.dbcp2.*** correspondientemente.
 - Ejemplo:
 - # Number of ms to wait before throwing an exception if no connection is available.
 - **spring.datasource.tomcat.max-wait=10000**
 - # Maximum number of active connections that can be allocated from this pool at the same time.
 - **spring.datasource.tomcat.max-active=50**
 - # Validate the connection before borrowing it from the pool.
 - **spring.datasource.tomcat.test-on-borrow=true**

Microservices



iii.v Spring Data JPA (i)

- **spring-boot-starter-data-jpa** provee las siguientes dependencias para trabajar con Spring Data JPA:
 - Hibernate: La implementación JPA más popular de la industria.
 - Spring Data JPA: Proyecto que implementa Repositorios y "queries" personalizados, basado en interfaces.
 - Spring ORM: Clases core adaptadoras para la integración de frameworks ORM.

Microservices



iii.v Spring Data JPA (j)

- Configuración Spring Data JPA con Spring Boot.
- Spring Boot no requiere del archivo "**persistence.xml**" para especificar las entidades JPA, en automático, Spring Boot escaneará, las entidades anotadas con **@Entity**, **@Embeddable** o **@MappedSuperclass**, desde el paquete base de la aplicación anotada con **@EnableAutoConfiguration** o **@SpringBootApplication**.
- Es posible mejorar el performance de la aplicación al hacer el escaneo de entidades mediante **@EntityScan**. Así mismo, si las entidades no están definidas bajo un sub-paquete del paquete base, es necesario especificar dicho paquete mediante **@EntityScan**.



iii.v Spring Data JPA (k)

- Definición de Spring Data JPA Repositories.
- Es posible definir un Repositorio genérico, agnóstico de la tecnología de persistencia mediante definir una interfaz que herede de la interfaz **Repository** o una de sus sub-interfaces.
- La interface **Repository** es una “**marker-interface**” y no define ningún método.
- Si se define un repositorio heredando de la interface **Repository**, es necesario definir los “**query methods**” a implementar por Spring Data JPA al vuelo.



iii.v Spring Data JPA (I)

- Definición de Spring Data JPA Repositories.

@Entity

```
public class City implements Serializable {
```

```
    @Id
```

```
    @GeneratedValue
```

```
    private Long id;
```

```
    @Column(nullable = false)
```

```
    private String name;
```

```
    @Column(nullable = false)
```

```
    private String state;
```

```
    ...
```

```
}
```

```
public interface CityRepository extends Repository<City, Long> {
```

```
    Page<City> findAll(Pageable pageable);
```

```
    City findByNameAndStateIgnoringCase(  
        String name, String state);
```

```
}
```



iii.v Spring Data JPA (m)

- Definición de Spring Data JPA Repositories.
- Spring Data, provee interfaces **Repository** específicas de la tecnología de persistencia a utilizar, tal como **JpaRepository** y **MongoRepository**.
- Dichas interfaces heredan de **CrudRepository** o **PagingAndSortingRepository**, las cuales ofrecen capacidades superiores tales como implementación de operaciones CRUD o paginación de resultados “**out-of-the-box**”.

Microservices



iii.v Spring Data JPA (n)

- Configurando propiedades JPA.
- Por default, Spring Boot crea bases de datos únicamente si, la base de datos, es en memoria (H2, HSQL o Derby).
- Para crear y eliminar una base de datos mediante hibernate es posible especificarlo mediante la propiedad:
`spring.jpa.hibernate.ddl-auto=create-drop`

Microservices



iii.v Spring Data JPA (ñ)

- Configurando propiedades JPA.
- Internamente Hibernate tiene una propiedad para crear y eliminar la base de datos **hibernate.hbm2ddl.auto**, sin embargo, para utilizar propiedades específicas de Hibernate es necesario hacerlo mediante el prefijo **spring.jpa.properties.***, ejemplo:
spring.jpa.properties.hibernate.hbm2ddl.auto
- Se recomienda siempre utilizar la propiedad **spring.jpa.hibernate.ddl-auto** para configurar cómo se creará la base de datos debido a que, dicha propiedad, tiene distintos valores default dependiendo de las condiciones en la ejecución de la aplicación.



iii.v Spring Data JPA (o)

- Configurando propiedades JPA.
- Si la base de datos de la aplicación es embebida, el valor de la propiedad **spring.jpa.hibernate.ddl-auto** será **"create-drop"**, para todos los demás caso es **"none"**.
- Atención de no eliminar accidentalmente la base de datos en producción.
- Para más opciones de auto-configuración de Spring Data JPA mediante Spring Boot:

<https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html>



iii.v Spring Data JPA (p)

- Configurando el patrón “Open EntityManager in View” (Open session in view).
- Si se está desarrollando una aplicación web, por default, Spring Boot registra el bean **OpenEntityManagerInViewInterceptor** el cual aplica el patrón “**Open EntityManager in View**” el cuál habilita la extracción de referencias **Lazy** sobre vistas web.
- Si no se requiere dicho comportamiento, es posible deshabilitarlo mediante la propiedad **spring.jpa.open-in-view=false** en el **application.properties**.



iii.v Spring Data JPA (q)

- Consola web para H2.
- La base de datos en memoria H2, provee una consola basada en web. Spring Boot auto-configura la consola si ocurren las siguientes condiciones:
 - Se está desarrollando una aplicación web.
 - La base de datos en memoria es H2 y el paquete `com.h2database:h2` está en el classpath.
 - Se está utilizando Spring Boot DevTools.

Microservices



iii.v Spring Data JPA (r)

- Consola web para H2.
- Si no se está utilizando Spring Boot DevTools, es posible habilitar la consola web H2 mediante la propiedad:
spring.h2.console.enabled=true.
- Por default, Spring Boot habilita la consola web H2 sobre la URL de la aplicación web **/h2-console**. Es posible personalizar el path de la consola web H2 mediante la propiedad **spring.h2.console.path**.



iii.v Spring Data JPA (s)

- **Práctica 5. Spring MVC y Spring Data JPA**
- Ingresar a la ruta: **{tu-workspace}/5-Spring-MVC-and-Spring-Data-JPA**
- Define la entidad **User** en el paquete **com.truper.springboot.practica5.entities**.
- Define el repositorio **UserRepository** heredando de **JpaRepository**. Define un método que busque un listado de entidades **User** a partir del nombre.
- En el paquete **com.truper.springboot.practica5.services.impl**, implementa la interface **IUserService**.



iii.v Spring Data JPA (t)

- **Práctica 5. Spring MVC y Spring Data JPA**
- Revisa los “**templates**” de Thymeleaf, de la ubicación **/src/main/resources/templates** y analiza qué objetos son requeridos para alimentar dichas vistas.
- Revisa también el archivo **data.sql** que está sobre raíz del classpath.
- Analiza dónde se encuentra el **favicon.ico** personalizado.
- En el paquete **com.truper.springboot.practica5.controller** implementa el controlador **UserController**, tratando de implementar las rutas y objetos requeridos para las vistas correspondientes.



iii.v Spring Data JPA (u)

- **Práctica 5. Spring MVC y Spring Data JPA**
- No es necesario realizar cambios a las vistas/templates HTML de Thymeleaf.
- Habilita la consola web de H2 sobre el path **/h2**.
- Ejecuta la clase principal, anotada con **@SpringBootApplication** y prueba tu trabajo en el navegador: <http://localhost:8080>

Microservices



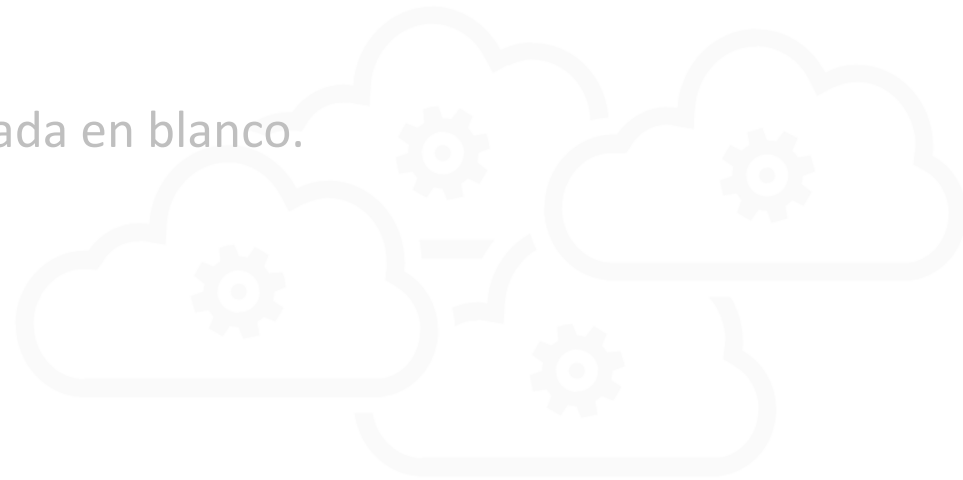
Resumen de la lección

iii.v Spring Data JPA

- Comprendimos como Spring Boot habilita la auto-configuración en aplicaciones Spring Data JPA.
- Comprendimos varios puntos importantes a considerar al utilizar la convención sobre configuración que implementa Spring Boot.
- Revisamos que no es necesario configurar ningún bean para implementar acceso a datos mediante Spring Data JPA y base de datos en memoria, útil para ambientes de desarrollo.
- Practicamos la experiencia adquirida en implementar una aplicación CRUD básica mediante Spring MVC y Spring Data JPA, mediante Spring Boot.



Esta página fue intencionalmente dejada en blanco.



Microservices



iii. Fundamentos Spring Boot 2.x

- iii.i Introducción a Spring Boot
- iii.ii Configuración de propiedades en Spring Boot
- iii.iii Perfiles
- iii.iv Spring MVC
- iii.v Spring Data JPA
- iii.vi Spring Data REST**
- iii.vii Spring Boot Actuator

Microservices



iii.vi Spring Data REST



Microservices



Objetivos de la lección

iii.vi Spring Data REST

- Conocer la auto-configuración de Spring Data REST mediante Spring Boot.
- Revisar a grandes rasgos las características que habilita Spring Boot a las aplicaciones Spring Data REST.
- Conocer como Spring Data REST implementa HATEOAS mediante HAL.
- Exponer repositorios Spring Data JPA como servicios REST mediante Spring Data REST sin mayor esfuerzo.
- Utilizar Swagger como plataforma de documentación de servicios e interface para probarlos.



iii.vi Spring Data REST (a)

- Spring Data REST expone los repositorios definidos mediante la interfaz **Repository**, de Spring Data, o cualquiera de sus sub-interfaces, como REST endpoints listos para ser consumidos por un cliente REST.
- Spring Boot habilita el desarrollo rápido REST endpoints con Spring Data REST mediante el módulo **"starter" spring-boot-starter-data-rest**, para su uso en conjunto con algún módulo soportado de Spring Data.
- Spring Data REST es soportado oficialmente por los módulos:
 - Spring Data JPA
 - Spring Data MongoDB
 - Spring Data Neo4j
 - Spring Data GemFire
 - Spring Data Cassandra



iii.vi Spring Data REST (b)

- Spring Data REST se construye encima de repositorios definidos mediante Spring Data y los exporta como REST endpoints.
- Spring Data REST expone como APIs REST, todos los repositorios **Repository** definidos o de alguna de sus sub-interfaces.
- Spring Boot expone una serie de propiedades útiles para configurar los repositorios REST expuestos como endpoints mediante el conjunto de propiedades: **spring.data.rest.***.
- No es requerida configuración adicional sobre Spring Boot.



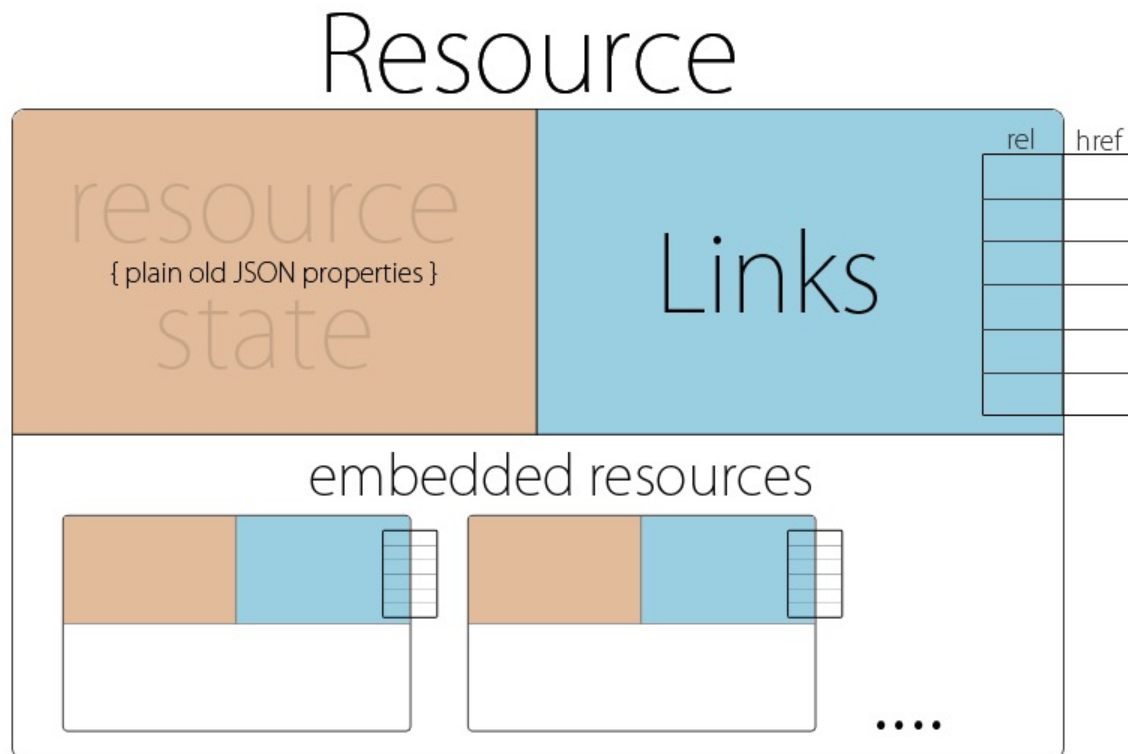
iii.vi Spring Data REST (c)

- Spring Data REST se apalanca mediante el uso correcto de HATEOAS, lo cual permite a los clientes REST, conocer las funcionalidades expuestas por las APIs REST que Spring Data REST expone.
- El principio básico de HATEOAS es que, los clientes puedan descubrir los servicios, es decir, que los servicios sean detectables a través de links.
- Existen algunos mecanismos de-facto, para representar links en formato JSON, Spring Data REST utiliza HAL para representar las respuestas que devuelven los servicios expuestos.



iii.vi Spring Data REST (d)

- HAL



<https://tools.ietf.org/html/draft-kelly-json-hal-08>



iii.vi Spring Data REST (e)

- Ejemplo representación HAL:

GET /orders/523 HTTP/1.1

Host: example.org

Accept: application/hal+json

HTTP/1.1 200 OK

Content-Type: application/hal+json

```
{
  "_links": {
    "self": { "href": "/orders/523" },
    "warehouse": { "href": "/warehouse/56" },
    "invoice": { "href": "/invoices/873" }
  },
  "currency": "USD",
  "status": "shipped",
  "total": 10.20
}
```

GET /orders HTTP/1.1

Host: example.org

Accept: application/hal+json

HTTP/1.1 200 OK

Content-Type: application/hal+json

```
{
  "_links": {
    "self": { "href": "/orders" },
    "next": { "href": "/orders?page=2" },
    "find": { "href": "/orders/{id}", "templated": true }
  },
  "_embedded": {
    "orders": [ {
      "_links": {
        "self": { "href": "/orders/123" },
        "basket": { "href": "/baskets/98712" },
        "customer": { "href": "/customers/7809" }
      },
      "total": 30.00,
      "currency": "USD",
      "status": "shipped"
    }, ... ]
  },
  "currentlyProcessing": 14,
  "shippedToday": 20
}
```



iii.vi Spring Data REST (f)

- Por default Spring Boot expone los REST endpoints, APIs REST, sobre la raíz de la aplicación web, o “**root**”.
- A partir de Spring Boot 1.2 es posible definir el path base desde donde se expondrán las APIs REST mediante la propiedad: **spring.data.rest.basePath**.
- Es posible definir la configuración de un recurso en particular mediante las anotaciones **@RepositoryRestResource** y **@RestResource**.
- La anotación **@CrossOrigin** sobre los repositorios expuestos habilitan el Cross-Domain.



iii.vi Spring Data REST (g)

- Ejemplo:

@CrossOrigin

```
@RepositoryRestResource(path = "usuarios",  
                           collectionResourceRel = "amigos",  
                           itemResourceRel = "my_self")  
public interface UserRepository extends JpaRepository<User, Long> {  
  
    @RestResource(exported = true, rel = "find", path = "find_by_name")  
    List<User> findByName(String name);  
}
```

Microservices



```
{
  "_embedded" : {
    "amigos" : [ {
      "name" : "Claudia",
      "email" : "claudia@boot.com",
      "id" : 1,
      "_links" : {
        "self" : {
          "href" : "http://localhost:8080/api-rest/usuarios/1"
        },
        "my_self" : {
          "href" : "http://localhost:8080/api-rest/usuarios/1"
        }
      }
    } ],
    {
      "name" : "Fernanda",
      "email" : "fernanda@boot.com",
      "id" : 2,
      "_links" : {
        "self" : {
          "href" : "http://localhost:8080/api-rest/usuarios/2"
        },
        "my_self" : {
          "href" : "http://localhost:8080/api-rest/usuarios/2"
        }
      }
    }
  ], { ... }, ... , { ... } ],
  ...
}
```

iii.vi Spring Data REST (h)

```
...
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/api-rest/usuarios{&sort}",
      "templated" : true
    },
    "profile" : {
      "href" : "http://localhost:8080/api-rest/profile/usuarios"
    },
    "search" : {
      "href" : "http://localhost:8080/api-rest/usuarios/search"
    }
  },
  "page" : {
    "size" : 100,
    "totalElements" : 11,
    "totalPages" : 1,
    "number" : 0
  }
}
```

GET /api-rest/usuarios HTTP/1.1
Host: localhost:8080
Accept: application/hal+json



iii.vi Spring Data REST (i)

GET /api-rest/**usuarios/search** HTTP/1.1

Host: localhost:8080

Accept: application/hal+json

HTTP/1.1 200 OK

Content-Type: application/hal+json

```
{
  "_links" : {
    "find" : {
      "href" : "http://localhost:8080/api-rest/usuarios/search/find_by_name{?name}",
      "templated" : true
    },
    "self" : {
      "href" : "http://localhost:8080/api-rest/usuarios/search"
    }
  }
}
```



GET /api-rest/usuarios/search/find_by_name?name=Fernanda HTTP/1.1

Host: localhost:8080

Accept: application/hal+json

HTTP/1.1 200 OK

Content-Type: application/hal+json

iii.vi Spring Data REST (j)

```
{
  "_embedded" : {
    "amigos" : [ {
      "name" : "Fernanda",
      "email" : "fernanda@boot.com",
      "id" : 2,
      "_links" : {
        "self" : {
          "href" : "http://localhost:8080/api-rest/usuarios/2"
        },
        "my_self" : {
          "href" : "http://localhost:8080/api-rest/usuarios/2"
        }
      }
    }
  ],
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/api-rest/usuarios/search/find_by_name?name=Fernanda"
    }
  }
}
```




iii.vi Spring Data REST (k)

- **Práctica 6. Spring Data REST**
- Ingresar a la ruta: **{tu-workspace}/6-Spring-Data-REST**
- La práctica **6-Spring-Data-REST** parte de la práctica anterior, **5-Spring-MVC-and-Spring-Data-JPA**. Se reutiliza la entidad **User** y se desechan el controlador **UserController**, la interface **IUserService** y su implementación **UserService**.
- A su vez, se elimina la dependencia **Thymeleaf**, debido a que la presente práctica expondrá el repositorio **UserRepository** como API REST mediante Spring Data REST.

Microservices



iii.vi Spring Data REST (I)

- **Práctica 6. Spring Data REST**
- Agregar la dependencia **spring-boot-starter-data-rest** al proyecto **6-Spring-Data-REST**.
- Analizar que se han agregado las dependencias **springfox-swagger2**, **springfox-data-rest**, **springfox-bean-validators** y **springfox-swagger-ui**, para la generación y auto-documentación de las APIs creadas por Spring Data REST con **swagger** (opcional).

Microservices



iii.vi Spring Data REST (m)

- **Práctica 6. Spring Data REST**
- Agregar la configuración necesaria para cumplir con las siguientes condiciones:
 - Que el servidor embebido se ejecute en el puerto **8080**.
 - Que la consola H2 este **habilitada**.
 - Que el path de la consola H2 sea **/h2**.
 - Que el path-base de los repositorios Spring Data REST estén definidos por **/api-rest**.
 - Que el tamaño de página por default de los repositorios expuestos por Spring Data REST sea de **5 elementos** embebidos.
- Se provee la configuración de Swagger en la clase **Swagger2Config** del paquete **com.truper.springboot.practica6.swaggerconfig**.



iii.vi Spring Data REST (n)

- **Práctica 6. Spring Data REST**
- Analizar que se han eliminado las dependencias de Thymeleaf en los archivos **index.html**, **add-user-form.html** y **update-user-form.html**. Dichos archivos contienen HTML plano.
- Ejecuta la clase principal, anotada con **@SpringBootApplication** y prueba tu trabajo en el navegador: <http://localhost:8080>
- Swagger UI se muestra en la URL: <http://localhost:8080/swagger-ui/>
- Analiza la funcionalidad implementada mediante JavaScript y los REST endpoints expuestos por Spring Data REST.



iii.vi Spring Data REST (ñ)

- **Práctica 6. Spring Data REST Client**
- Ingresar a la ruta: **{tu-workspace}/6-Spring-Data-REST-Client**
- La práctica **6-Spring-Data-REST-Client** parte de la práctica anterior, **6-Spring-Data-REST**. Se reutilizan las vistas HTML del proyecto anterior y se deshecha todo lo demás.
- El proyecto **6-Spring-Data-REST-Client** se desplegará en el puerto **9090** y consumirá las APIs expuestas por el proyecto **6-Spring-Data-REST** desplegado en el puerto **8080**.



iii.vi Spring Data REST (o)

- **Práctica 6. Spring Data REST Client**
- Si el proyecto **6-Spring-Data-REST** ha sido desplegado en un puerto diferente al **8080**, revisar y asignar correctamente la variable **USER_API_HOST** de los archivos **index.js**, **add-user.js** y **update-user.js** del proyecto **6-Spring-Data-REST-Client**.
- Confirmar que la configuración del proyecto **6-Spring-Data-REST-Client** defina que se desplegará en el puerto **9090**.

Microservices



iii.vi Spring Data REST (p)

- **Práctica 6. Spring Data REST Client**
- Ejecuta la clase principal, de ambos proyectos, anotada con **@SpringBootApplication** y prueba tu trabajo en el navegador:
<http://localhost:8080> (**6-Spring-Data-REST**) y
<http://localhost:9090> (**6-Spring-Data-REST-Client**).
- Analiza la funcionalidad implementada mediante JavaScript en el proyecto **6-Spring-Data-REST-Client**, el cual consume los REST endpoints expuestos por el proyecto **6-Spring-Data-REST**.
- ¿Por qué no funciona la aplicación **6-Spring-Data-REST-Client**? ¿Qué comenta la consola JS? Realiza las correcciones necesarias al proyecto **6-Spring-Data-REST**.



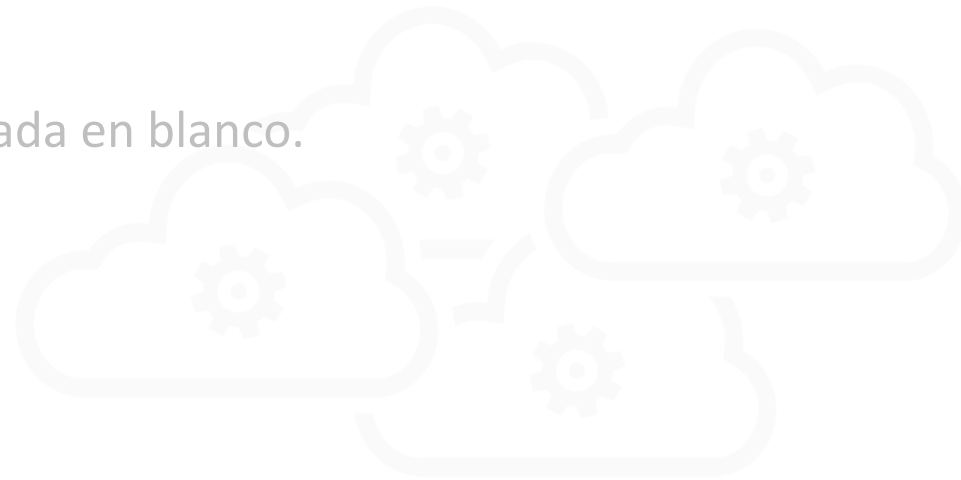
Resumen de la lección

iii.vi Spring Data REST

- Comprendimos el mecanismo de auto-configuración de Spring Data REST mediante Spring Boot.
- Implementamos servicios REST a partir de la definición de repositorios Spring Data JPA.
- Implementamos una aplicación simple que consuma dichos servicios REST a través de JavaScript, sin utilizar ninguna tecnología de presentación Java web del lado del servidor (no JSPs, no Thymeleaf).
- Implementamos una aplicación cliente que consuma los servicios REST expuestos bajo otro puerto mediante cross-domain.



Esta página fue intencionalmente dejada en blanco.



Microservices



iii. Fundamentos Spring Boot 2.x

- iii.i Introducción a Spring Boot
- iii.ii Configuración de propiedades en Spring Boot
- iii.iii Perfiles
- iii.iv Spring MVC
- iii.v Spring Data JPA
- iii.vi Spring Data REST
- iii.vii **Spring Boot Actuator**

Microservices



iii.vii Spring Boot Actuator



Microservices



Objetivos de la lección

iii.vii Spring Boot Actuator

- Conocer la auto-configuración de Spring Boot Actuator.
- Conocer las funcionalidades “**out-of-the-box**” que ofrece Spring Boot Actuator.
- Implementar métricas mediante Spring Boot Actuator.
- Comprender que son los actuadores o “**actuators**” e implementar un “actuator” personalizado.

Microservices



iii.vii Spring Boot Actuator (a)

- Entre todas las opciones de auto-configuración que provee Spring Boot en conjunción con los múltiples proyectos que soporta, Spring Boot ofrece “**Actuators**”.
- Spring Boot ofrece “**out-of-the-box**” funcionalidades adicionales que apoyan, a los desarrolladores y personal de operaciones, a monitorear y administrar la aplicación al momento de estar en producción.
- Es posible administrar y monitorear la aplicación mediante “**endpoints**” HTTP o a través de JMX.



iii.vii Spring Boot Actuator (b)

- Spring Boot Actuator habilita la administración y monitorización de las aplicaciones mediante el módulo “**starter**” **spring-boot-starter-actuator**.
- Un “**actuator**” o actuador, es un término proveniente de la industria manufacturera que refiere a un dispositivo mecánico para mover o controlar algo. Los actuadores pueden generar una gran cantidad de movimiento a partir de un ligero cambio.
- Los actuadores o “**actuators**” se disponibilizan a través de “**endpoints**” HTTP o mediante JMX.



iii.vii Spring Boot Actuator (c)

- Spring Boot Actuator incluye varios **"actuators"** de entre los cuáles destacan:
 - beans
 - env
 - health
 - httptrace
 - info
 - metrics
 - mappings
 - shutdown, entre otros.

<https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-endpoints>



iii.vii Spring Boot Actuator (d)

- Por default, todos los endpoints "**actuators**" están habilitados por default, excepto el endpoint "**shutdown**".
- Se utiliza la propiedad **management.endpoint.<endpoint>.enabled** para habilitar un endpoint.
- Si se desea, es posible deshabilitar todos los endpoints por default, y habilitar sólo aquellos requeridos para un momento determinado por la aplicación mediante la anotación:
management.endpoints.enabled-by-default



iii.vii Spring Boot Actuator (e)

- Los endpoints se configuran, uno a uno mediante la propiedad **management.endpoint.<nombre-del-endpoint>**.
- Por default, Spring Boot Actuator, habilita los endpoints a través de hipermedia, mediante un endpoint especial para el auto-descubrimiento de actuators bajo el path base **/actuator**. Es posible configurar éste path base mediante la propiedad **management.endpoints.web.base-path**.

Microservices



iii.vii Spring Boot Actuator (f)

- Los endpoints configurados con Spring Boot Actuator, están deshabilitados para el uso “**cross-domain**” de los mismos; para habilitarlos es necesario configurar las siguientes propiedades:
management.endpoints.web.cors.allowed-origins
management.endpoints.web.cors.allowed-methods

Microservices



iii.vii Spring Boot Actuator (g)

- Actuadores personalizados.
- Para definir actuadores personalizados utilizamos la anotación **@Endpoint**, la cuál expone la clase anotada como un endpoint la cual deberá responder a solicitudes **GET, POST y/o DELETE** para manipular la información que el endpoint requiera.
- Las anotaciones **@ReadOperation**, **@WriteOperation** y **@DeleteOperation**, sobre métodos de una clase anotada con **@Endpoint** expondrán dicha funcionalidad mediante el verbo correspondiente HTTP.



iii.vii Spring Boot Actuator (h)

- Recibir parámetros sobre actuadores personalizados.
- El paso de parámetros a los métodos “**operation**”, aquellos anotados con **@ReadOperation**, **@WriteOperation** y **@DeleteOperation**, se da mediante el nombre de sus parámetros y el nombre de los atributos de las llaves de “**query-parameters**” o el nombre de las llaves del objeto JSON de entrada en la petición HTTP.

Microservices



iii.vii Spring Boot Actuator (i)

- Recibir parámetros sobre actuadores personalizados.

@Component

@Endpoint(id = "features")

public class FeaturesEndpoint {

@Autowired

private Map<String, Feature> features;

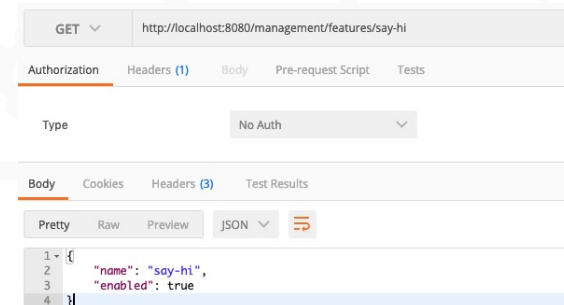
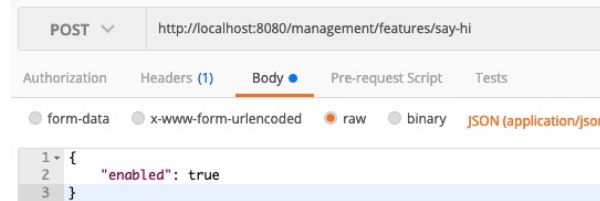
@ReadOperation

```
public Feature feature(@Selector String name) {
    return features.get(name);
}
```

@WriteOperation

```
public void configureFeature(@Selector String name, boolean enabled) {
    features.put(name, Feature.builder().name(name).enabled(enabled).build());
}
...

```





iii.vii Spring Boot Actuator (j)

- De forma análoga, mediante las anotaciones **@ControllerEndpoint** y **@RestControllerEndpoint**, es posible definir endpoints actuadores, definidos mediante anotaciones de Spring MVC o Spring WebFlux.
- No se revisa Spring Boot Actuator en su extensión debido a que no es objetivo del curso.



iii.vii Spring Boot Actuator (k)

- **Práctica 7. Spring Boot Actuator**
- Ingresar a la ruta: **{tu-workspace}/7-Spring-Boot-Actuator**
- Habilita la exposición web, es decir, vía HTTP de los siguientes **“actuators”**: **shutdown, health, info, beans, env, metrics y features**, mediante la propiedad **management.endpoints.web.exposure.include**.
- Habilita el **“actuator” shutdown**, para apagar la aplicación web correctamente.
- Cambia el path base de Spring Boot Actuator de: **/actuator** a **/management**.



iii.vii Spring Boot Actuator (I)

- **Práctica 7. Spring Boot Actuator**
- Analiza la definición del bean **features** en la clase principal del proyecto.
- Analiza la clase **Feature**.
- Analiza la clase de configuración **CustomFeaturesConfig**.
- Opcional. Analiza las clases del paquete **com.truper.springboot.practica7.spel**.
- Analiza la clase **HelloController**.



iii.vii Spring Boot Actuator (m)

- **Práctica 7. Spring Boot Actuator**
- Implementa la clase **@Endpoint FeaturesEndpoint** para definir métodos de lectura (dos, uno que resuelva todas las features y otro que resuelva por nombre de feature), escritura (para definir un nuevo feature) y de borrado (para eliminar una feature).
- Ejecuta la clase principal del proyecto, anotada con **@SpringBootApplication** y prueba tu trabajo en el navegador: <http://localhost:8080>. Accede al path **/management** y manipula los diversos endpoints expuestos por Spring Boot Actuator.



iii.vii Spring Boot Actuator (n)

- **Práctica 7. Spring Boot Actuator**
- Prueba el endpoint **“/management/health”** ¿Qué se visualiza? Extiende la visualización del endpoint **“health”** mediante la propiedad **management.endpoint.health.show-details**.
- Prueba el endpoint **“/management/info”** ¿Qué se visualiza? Extiende la visualización del endpoint **“info”** mediante definir propiedades **info.***.
- Agrega la dependencia Spring Data REST HAL Browser:
org.springframework.data:spring-data-rest-hal-browser, vuelve a lanzar la clase principal y accede a: <http://localhost:8080/browser/index.html>



Resumen de la lección

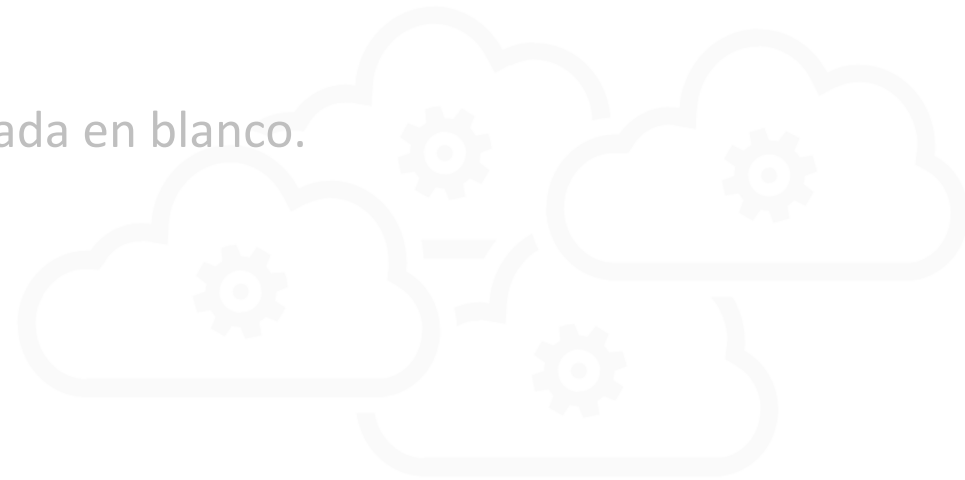
iii.vii Spring Boot Actuator

- Comprendimos algunas de las funcionalidades **“out-of-the-box”** que ofrece Spring Boot Actuator.
- Aprendimos que es un “Actuator” o actuador y como beneficia a nuestra aplicación para la administración y monitoreo
- Implementamos métricas y actuadores personalizados.
- Analizamos algunos de los “Actuator” endpoints más utilizados por default en aplicaciones Spring Boot.

Microservices



Esta página fue intencionalmente dejada en blanco.



Microservices