



EBook Gratis

APRENDIZAJE spring-boot

Free unaffiliated eBook created from
Stack Overflow contributors.

#spring-
boot

Tabla de contenido

Acerca de.....	1
Capítulo 1: Comenzando con el arranque de primavera.....	2
Observaciones.....	2
Versiones.....	2
Examples.....	2
Instalación o configuración.....	2
Aplicación web Spring Boot simple que usa Gradle como sistema de compilación.....	4
Capítulo 2: Almacenamiento en caché con Redis usando Spring Boot para MongoDB.....	6
Examples.....	6
¿Por qué caching?.....	6
El sistema basico.....	6
Capítulo 3: Aplicación web de arranque Spring-Responsive con JHipster.....	13
Examples.....	13
Crea la aplicación Spring Boot usando jHipster en Mac OS.....	13
Capítulo 4: Bota de primavera + JPA + mongoDB.....	17
Examples.....	17
Operación CRUD en MongoDB usando JPA.....	17
Controlador del cliente.....	18
Repositorio de clientes.....	19
pom.xml.....	19
Insertar datos usando rest client: método POST.....	19
Obtener URL de solicitud.....	20
Obtener el resultado de la solicitud:.....	20
Capítulo 5: Bota de Primavera + JPA + RESTO.....	22
Observaciones.....	22
Examples.....	22
Arranque de arranque de primavera.....	22
Objeto de dominio.....	22
Interfaz de repositorio.....	23
Configuración de Maven.....	24

Capítulo 6: Conectando una aplicación de arranque de primavera a MySQL	26
Introducción	26
Observaciones	26
Examples	26
Ejemplo de arranque de primavera usando MySQL	26
Capítulo 7: Controladores	31
Introducción	31
Examples	31
Controlador de reposacabezas de muelle	31
Capítulo 8: Creación y uso de múltiples archivos de aplicaciones.propiedades	34
Examples	34
Dev y Prod entorno utilizando diferentes fuentes de datos	34
Establezca el perfil de resorte correcto construyendo la aplicación automáticamente (maven	35
Capítulo 9: Escaneo de paquetes	38
Introducción	38
Parámetros	38
Examples	39
@SpringBootApplication	39
@ComponentScan	40
Creando tu propia autoconfiguración	41
Capítulo 10: Implementación de la aplicación de ejemplo utilizando Spring-boot en Amazon E	43
Examples	43
Implementación de una aplicación de ejemplo utilizando Spring-boot en formato Jar en AWS	43
Capítulo 11: Instalación de la CLI de inicio de Spring	50
Introducción	50
Observaciones	50
Examples	51
Instalación manual	51
Instalar en Mac OSX con HomeBrew	51
Instalar en Mac OSX con MacPorts	51
Instalar en cualquier sistema operativo con SDKMAN!	51

Capítulo 12: Microservicio de arranque de primavera con JPA	52
Examples	52
Clase de aplicación	52
Modelo de libro	52
Repositorio de libros	53
Habilitar la validación	53
Cargando algunos datos de prueba	54
Añadiendo el validador	54
Gradle Build File	55
Capítulo 13: Pruebas en Spring Boot	57
Examples	57
Cómo probar una aplicación de arranque de primavera simple	57
Cargando diferentes archivos yaml [o propiedades] o anular algunas propiedades	60
Cargando diferentes archivos yaml	60
Opciones alternativas	60
Capítulo 14: Servicios de descanso	62
Parámetros	62
Examples	62
Creación de un servicio REST	62
Creando un servicio de descanso con JERSEY y Spring Boot	65
1. Configuración del proyecto	65
2. Crear un controlador	65
Configuraciones de Jersey de cableado	66
4. Hecho	66
Consumir una API REST con RestTemplate (GET)	66
Capítulo 15: Spring boot + Hibernate + Web UI (Thymeleaf)	69
Introducción	69
Observaciones	69
Examples	69
Dependencias maven	69
Configuración de hibernación	70

Entidades y Repositorios.....	71
Recursos Thymeleaf y Spring Controller.....	71
Capítulo 16: Spring Boot + Spring Data Elasticsearch.....	73
Introducción.....	73
Examples.....	73
Integración de Spring Boot y Spring Data Elasticsearch.....	73
Integración elasticsearch de arranque y datos de primavera.....	73
Capítulo 17: Spring boot + Spring Data JPA.....	81
Introducción.....	81
Observaciones.....	81
Anotaciones.....	81
Documentacion oficial.....	81
Examples.....	82
Ejemplo básico de integración de Spring Boot y Spring Data JPA.....	82
Clase principal.....	82
Clase de entidad.....	82
Propiedades transitorias.....	83
Clase DAO.....	84
Clase de servicio.....	84
Servicio de frijol.....	85
Clase de controlador.....	86
Archivo de propiedades de la aplicación para la base de datos MySQL.....	87
Archivo SQL.....	87
archivo pom.xml.....	87
Construyendo un JAR ejecutable.....	88
Capítulo 18: Spring Boot- Hibernate-REST Integration.....	89
Examples.....	89
Añadir soporte Hibernate.....	89
Añadir soporte REST.....	90
Capítulo 19: Spring-Boot + JDBC.....	92
Introducción.....	92

Observaciones.....	92
Examples.....	93
archivo schema.sql.....	93
Primera aplicación de arranque JdbcTemplate.....	94
data.sql.....	94
Capítulo 20: ThreadPoolTaskExecutor: configuración y uso.....	95
Examples.....	95
configuración de la aplicación.....	95
Creditos.....	96

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: [spring-boot](#)

It is an unofficial and free spring-boot ebook created for educational purposes. All the content is extracted from [Stack Overflow Documentation](#), which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official spring-boot.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Comenzando con el arranque de primavera

Observaciones

Esta sección proporciona una descripción general de qué es spring-boot y por qué un desarrollador puede querer usarlo.

También debe mencionar cualquier tema importante dentro de Spring-Boot y vincular a los temas relacionados. Dado que la Documentación para spring-boot es nueva, es posible que deba crear versiones iniciales de esos temas relacionados.

Versiones

Versión	Fecha de lanzamiento
1.5	2017-01-30
1.4	2016-07-28
1.3	2015-11-16
1.2	2014-12-11
1.1	2014-06-10
1.0	2014-04-01

Examples

Instalación o configuración

La configuración con Spring Boot por primera vez es bastante rápida gracias al arduo trabajo de Spring Community.

Requisitos previos:

1. Java instalado
2. Se recomienda el IDE de Java no requerido (IntelliJ, Eclipse, Netbeans, etc.)

No es necesario tener instalado Maven y / o Gradle. Los proyectos generados por [Spring Initializr](#) vienen con un Maven Wrapper (comando `mvnw`) o Gradle Wrapper (comando `gradlew`).

Abra su navegador web en <https://start.spring.io> Este es un launchpad para crear nuevas

aplicaciones Spring Boot por ahora, lo haremos con el mínimo indispensable.

Siéntase libre de cambiar de Maven a Gradle si esa es su herramienta de construcción preferida.

Busque "Web" en "Buscar dependencias" y agréguela.

Haga clic en Generar proyecto!

Esto descargará un archivo zip llamado demo. Siéntase libre de extraer este archivo donde lo desee en su computadora.

Si selecciona maven, navegue en un indicador de comando hacia el directorio base y emita un comando `mvn clean install`

Debería obtener una salida de éxito de compilación:

```
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 3.908 s
[INFO] Finished at: 2016-07-21T07:15:06-04:00
[INFO] Final Memory: 27M/331M
[INFO] -----
C:\Users\gsd4tyk\Downloads\demo>
```

Ejecutando su aplicación: `mvn spring-boot:run`

Ahora su aplicación Spring Boot se inicia. Navegue por su navegador web a localhost: 8080

Felicidades Acabas de poner en marcha tu primera aplicación Spring Boot. Ahora vamos a agregar un poquito de código para que puedas verlo funcionar.

Así que usa `ctrl + c` para salir de tu servidor actual en ejecución.

Vaya a: `src/main/java/com/example/DemoApplication.java` Actualice esta clase para tener un controlador

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@SpringBootApplication
public class DemoApplication {

    @RequestMapping("/")
    String home() {
        return "Hello World!";
    }

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

```
}  
}
```

Lo bueno ahora permite construir y ejecutar el proyecto nuevamente con `mvn clean install`
`spring-boot:run` !

Ahora navegue su navegador web a localhost: 8080

Hola Mundo!

Felicidades Acabamos de completar la creación de una aplicación Spring Boot y configuramos nuestro primer controlador para que devuelva "¡Hola mundo!" ¡Bienvenido al mundo de Spring Boot!

Aplicación web Spring Boot simple que usa Gradle como sistema de compilación.

Este ejemplo asume que ya ha instalado Java y [Gradle](#) .

Utilice la siguiente estructura de proyecto:

```
src/  
  main/  
    java/  
      com/  
        example/  
          Application.java  
build.gradle
```

`build.gradle` es su script de compilación para el sistema de compilación Gradle con el siguiente contenido:

```
buildscript {  
    ext {  
        //Always replace with latest version available at http://projects.spring.io/spring-  
boot/#quick-start  
        springBootVersion = '1.5.6.RELEASE'  
    }  
    repositories {  
        jcenter()  
    }  
    dependencies {  
        classpath("org.springframework.boot:spring-boot-gradle-plugin:${springBootVersion}")  
    }  
}  
  
apply plugin: 'java'  
apply plugin: 'org.springframework.boot'  
  
repositories {  
    jcenter()  
}  
  
dependencies {
```

```
compile('org.springframework.boot:spring-boot-starter-web')
}
```

`Application.java` es la clase principal de la aplicación web Spring Boot:

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication // same as @Configuration @EnableAutoConfiguration @ComponentScan
@RestController
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class);
    }

    @RequestMapping("/hello")
    private String hello() {
        return "Hello World!";
    }
}
```

Ahora puedes ejecutar la aplicación web Spring Boot con

```
gradle bootRun
```

y acceda al punto final HTTP publicado usando `curl`

```
curl http://localhost:8080/hello
```

o su navegador abriendo [localhost: 8080 / hello](http://localhost:8080/hello) .

Lea Comenzando con el arranque de primavera en línea: <https://riptutorial.com/es/spring-boot/topic/829/comenzando-con-el-arranque-de-primavera>

Capítulo 2: Almacenamiento en caché con Redis usando Spring Boot para MongoDB

Examples

¿Por qué caching?

Hoy en día, el rendimiento es una de las métricas más importantes que debemos evaluar al desarrollar un servicio / aplicación web. Mantener a los clientes comprometidos es fundamental para cualquier producto y, por esta razón, es extremadamente importante mejorar el rendimiento y reducir los tiempos de carga de la página.

Al ejecutar un servidor web que interactúa con una base de datos, sus operaciones pueden convertirse en un cuello de botella. MongoDB no es una excepción aquí y, a medida que nuestra base de datos MongoDB se amplía, las cosas realmente pueden disminuir. Este problema puede empeorar incluso si el servidor de la base de datos se separa del servidor web. En tales sistemas, la comunicación con la base de datos puede causar una gran sobrecarga.

Afortunadamente, podemos usar un método llamado almacenamiento en caché para acelerar las cosas. En este ejemplo, presentaremos este método y veremos cómo podemos usarlo para mejorar el rendimiento de nuestra aplicación utilizando Spring Cache, Spring Data y Redis.

El sistema básico

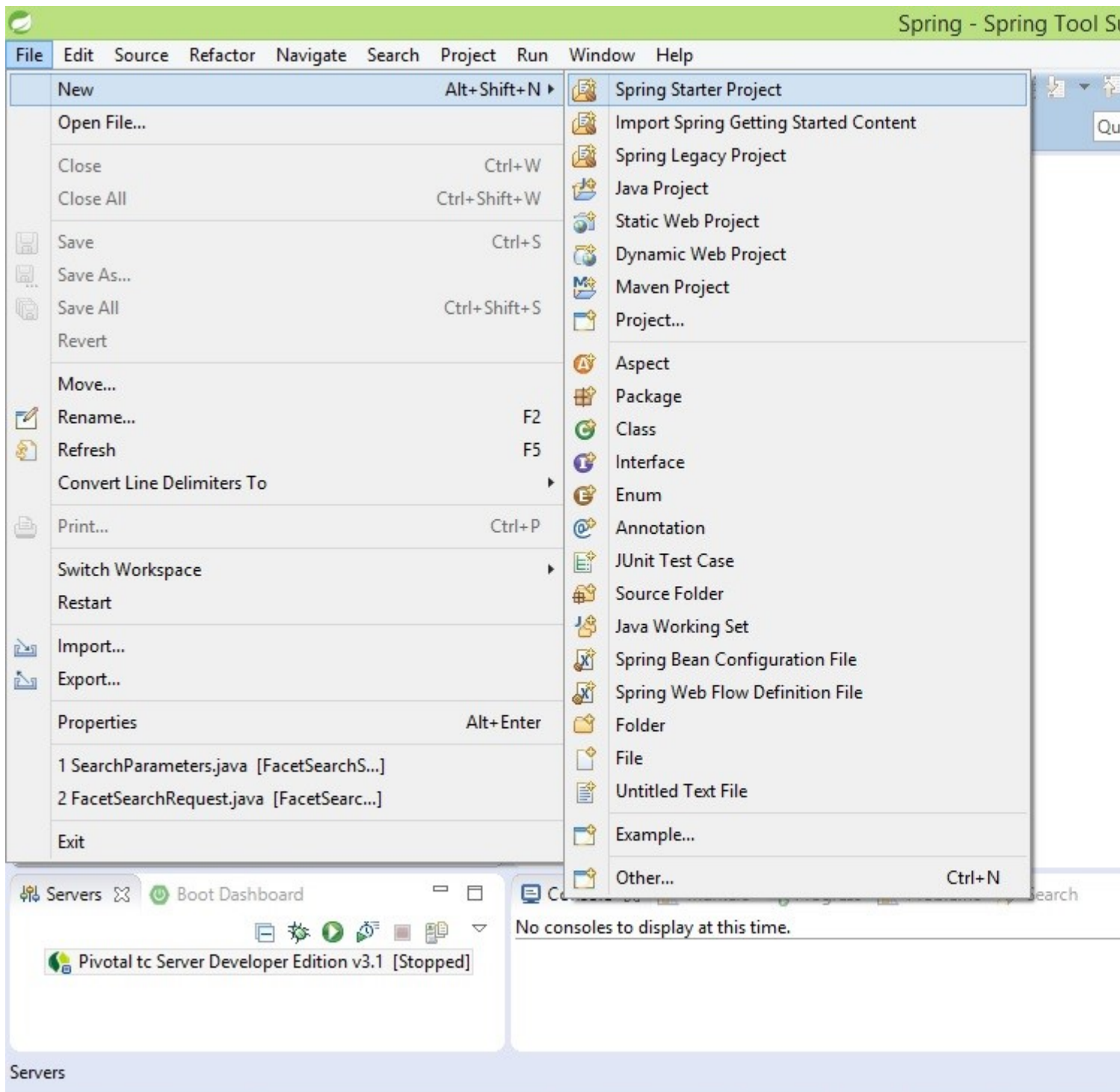
Como primer paso, construiremos un servidor web básico que almacena datos en MongoDB. Para esta demostración, lo llamaremos "Biblioteca rápida". El servidor tendrá dos operaciones básicas:

`POST /book` : este punto final recibirá el título, el autor y el contenido del libro, y creará una entrada de libro en la base de datos.

`GET /book/ {title}` : este punto final obtendrá un título y devolverá su contenido. Asumimos que los títulos identifican de forma única los libros (por lo tanto, no habrá dos libros con el mismo título). Una mejor alternativa sería, por supuesto, usar una identificación. Sin embargo, para mantener las cosas simples, simplemente usaremos el título.

Este es un sistema de biblioteca simple, pero agregaremos habilidades más avanzadas más adelante.

Ahora, vamos a crear el proyecto utilizando Spring Tool Suite (compilación con eclipse) y el proyecto de inicio de primavera



Estamos construyendo nuestro proyecto usando Java y para construir estamos usando maven, seleccionamos valores y hacemos clic en siguiente

New Spring Starter Project

Name:

☒ Use default location

Location:

Type: Packaging:

Java Version: Language:

Group:

Artifact:

Version:

Description:


Package:

Working sets

☐ Add project to working sets

Working sets:

Seleccione MongoDB, Redis desde NOSQL y Web desde el módulo web y haga clic en Finalizar. Estamos utilizando Lombok para la generación automática de Setters y getters de valores de modelo, por lo que necesitamos agregar la dependencia de Lombok al POM



New Spring Starter Project

☐ Security

☐ Cache

☐ Retry

☐ AOP

☐ DevTools

☐ Lombok

☐ Atomikos (JTA)

☐ Validation

☐ Bitronix (JTA)

☐ Session

☐ Batch

☐ JMS (HornetQ)

☐ Integration

☐ AMQP

☐ Activiti

☐ Mail

☐ JMS (Artemis)

☒ MongoDB

☒ Redis

☐ Cassandra

☐ Gemfire

☐ Couchbase

☐ Solr

☐ Redis

☐ Elasticsearch

☐ Actuator

☐ Actuator Docs

☐ Remote Shell

☐ JPA

☐ HSQLDB

☐ JOOQ

☐ Apache Derby

☐ JDBC

☐ MySQL

☐ H2

☐ PostgreSQL

☐ Facebook

☐ LinkedIn

☐ Twitter

☐ Freemarker

☐ Mustache

☐ Velocity

☐ Groovy Templates

☐ Thymeleaf

☒ Web

☐ Ratpack

☐ Rest Repositories HAL Browser

☐ Websocket

☐ Vaadin

☐ Mobile


☐ WS

☐ Rest Repositories

☐ REST Docs

☐ Jersey (JAX-RS)

☐ HATEOAS



< Back

Next >

Finish

https://riptutorial.com/es/home

9


```

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb</artifactId>
  </dependency>
  <dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>

```

MongoDbRedisCacheApplication.java contiene el método principal que se utiliza para ejecutar la aplicación Spring Boot add

Cree un modelo de libro de clase que contenga id, título del libro, autor, descripción y anote con @Data para generar configuradores automáticos y captadores del proyecto jar lombok

```

package com.example;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.index.Indexed;
import lombok.Data;
@Data
public class Book {
    @Id
    private String id;
    @Indexed
    private String title;
    private String author;
    private String description;
}

```

Spring Data crea todas las operaciones básicas de CRUD para nosotros automáticamente, así que vamos a crear BookRepository.Java que encuentra libro por título y elimina libro

```

package com.example;
import org.springframework.data.mongodb.repository.MongoRepository;
public interface BookRepository extends MongoRepository<Book, String>
{
    Book findByTitle(String title);
    void delete(String title);
}

```


Vamos a crear `WebServicesController` que guarda los datos en MongoDB y recuperamos los datos por `idTitle` (título de la cadena de variable de la variable de usuario).

```
package com.example;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;
@RestController
public class WebServicesController {
    @Autowired
    BookRepository repository;
    @Autowired
    MongoTemplate mongoTemplate;
    @RequestMapping(value = "/book", method = RequestMethod.POST)
    public Book saveBook(Book book)
    {
        return repository.save(book);
    }
    @RequestMapping(value = "/book/{title}", method = RequestMethod.GET)
    public Book findBookByTitle(@PathVariable String title)
    {
        Book insertedBook = repository.findByTitle(title);
        return insertedBook;
    }
}
```

Agregar el caché Hasta ahora hemos creado un servicio web de biblioteca básico, pero no es sorprendentemente rápido. En esta sección, intentaremos optimizar el método `findBookByTitle ()` almacenando en caché los resultados.

Para tener una mejor idea de cómo lograremos este objetivo, volvamos al ejemplo de las personas sentadas en una biblioteca tradicional. Digamos que quieren encontrar el libro con un título determinado. En primer lugar, mirarán alrededor de la mesa para ver si ya la han llevado allí. Si lo han hecho, eso es genial! Acaban de tener un hit de caché que está encontrando un elemento en el caché. Si no lo encontraron, tuvieron una falla de caché, lo que significa que no encontraron el elemento en el caché. En el caso de que falte un artículo, tendrán que buscar el libro en la biblioteca. Cuando lo encuentren, lo mantendrán en su mesa o lo insertarán en el caché.

En nuestro ejemplo, seguiremos exactamente el mismo algoritmo para el método `findBookByTitle ()`. Cuando se le solicite un libro con un título determinado, lo buscaremos en el caché. Si no lo encuentra, lo buscaremos en el almacenamiento principal, que es nuestra base de datos MongoDB.

Usando redis

Agregar `spring-boot-data-redis` a nuestra ruta de clase permitirá que Spring Boot realice su magia. Creará todas las operaciones necesarias mediante la configuración automática.

Anotemos ahora el método con la siguiente línea para almacenar en caché y dejar que Spring

Boot haga su magia

```
@Cacheable (value = "book", key = "#title")
```

Para eliminar de la memoria caché cuando se elimina un registro, simplemente haga una anotación con la línea inferior en BookRepository y deje que Spring Boot maneje la eliminación de la memoria caché por nosotros.

```
@CacheEvict (value = "book", key = "#title")
```

Para actualizar los datos, debemos agregar la siguiente línea al método y dejar que Spring Boot se maneje

```
@CachePut (value = "book", key = "#title")
```

Puedes encontrar el código completo del proyecto en [GitHub](#)

Lea Almacenamiento en caché con Redis usando Spring Boot para MongoDB en línea:

<https://riptutorial.com/es/spring-boot/topic/6496/almacenamiento-en-cache-con-redis-usando-spring-boot-para-mongodb>

Capítulo 3: Aplicación web de arranque Spring-Responsive con JHipster

Examples

Crea la aplicación Spring Boot usando jHipster en Mac OS

jHipster le permite iniciar una aplicación web Spring Boot con un back-end API REST y un front-end AngularJS y Twitter Bootstrap.

Más sobre jHipster aquí: [Documentación jHipster](#)

Instalar cerveza:

```
/usr/bin/ruby -e "$(curl -fsSL  
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Ver información adicional sobre cómo instalar brew aquí: [Instalar Brew](#)

Instalar gradle

Gradle es un sistema de gestión y construcción de dependencias.

```
brew install gradle
```

Instalar git

Git es una herramienta de control de versiones.

```
brew install git
```

Instala NodeJS

NodeJS le da acceso a npm, el administrador de paquetes de nodos que se necesita para instalar otras herramientas.

```
brew install node
```

Instalar Yeoman

Yeoman es un generador

```
npm install -g yo
```

Instalar Bower

Bower es una herramienta de gestión de dependencias.

```
npm install -g bower
```

Instalar Gulp

Gulp es un corredor de tareas

```
npm install -g gulp
```

Instalar jHipster Yeoman Generator

Este es el generador jHipster.

```
npm install -g generator-jhipster
```

Crear una aplicación

Abra una ventana de terminal.

Navigate hasta el directorio raíz donde guardará sus proyectos. Crea un directorio vacío en el que crearás tu aplicación.

```
mkdir myapplication
```

Ir a ese directorio

```
cd myapplication/
```

Para generar su aplicación, escriba

```
yo jhipster
```

Se te preguntarán las siguientes preguntas.

¿Qué tipo de aplicación te gustaría crear?

Su tipo de aplicación depende de si desea utilizar una arquitectura de microservicios o no. Una explicación completa sobre microservicios está disponible aquí, si no está seguro, use la "aplicación monolítica" predeterminada.

Elija la *aplicación Monolítica* por defecto si no está seguro

¿Cuál es tu nombre de paquete Java predeterminado?

Su aplicación Java utilizará esto como su paquete raíz.

¿Qué tipo de autenticación le gustaría usar?

Use la *seguridad* básica de *Spring* basada en sesión de manera predeterminada si no está seguro

¿Qué tipo de base de datos le gustaría usar?

¿Qué base de datos de desarrollo te gustaría usar?

Esta es la base de datos que utilizará con su perfil de "desarrollo". Usted puede utilizar:

Usa H2 por defecto si no estás seguro

H2, corriendo en memoria. Esta es la forma más fácil de usar JHipster, pero sus datos se perderán cuando reinicie su servidor.

¿Quieres usar la caché de segundo nivel de Hibernate?

Hibernate es el proveedor de JPA utilizado por JHipster. Por motivos de rendimiento, le recomendamos encarecidamente que utilice un caché y que lo sintonice de acuerdo con las necesidades de su aplicación. Si elige hacerlo, puede usar ehcache (caché local) o Hazelcast (caché distribuido, para usar en un entorno agrupado)

¿Quieres utilizar un motor de búsqueda en tu aplicación? Elasticsearch se configurará utilizando Spring Data Elasticsearch. Puedes encontrar más información en nuestra guía de Elasticsearch.

Elige no si no estás seguro.

¿Quieres usar sesiones HTTP agrupadas?

De forma predeterminada, JHipster usa una sesión HTTP solo para almacenar la información de autenticación y autorizaciones de Spring Security. Por supuesto, puede optar por poner más datos en sus sesiones HTTP. El uso de sesiones HTTP causará problemas si está ejecutando en un clúster, especialmente si no usa un equilibrador de carga con "sesiones pegajosas". Si desea replicar sus sesiones dentro de su grupo, elija esta opción para tener configurado Hazelcast.

Elige no si no estás seguro.

¿Quieres usar WebSockets? Websockets se puede habilitar usando Spring Websocket. También proporcionamos una muestra completa para mostrarle cómo usar el marco de manera eficiente.

Elige no si no estás seguro.

¿Te gustaría usar Maven o Gradle? Puedes construir tu aplicación Java generada con Maven o Gradle. Maven es más estable y más maduro. Gradle es más flexible, más fácil de extender y más exageración.

Elija *Gradle* si no está seguro

¿Le gustaría usar el preprocesador de hojas de estilo LibSass para su CSS? Node-sass una gran solución para simplificar el diseño de CSS. Para ser utilizado de manera eficiente, deberá ejecutar un servidor Gulp, que se configurará automáticamente.

Elige no si no estás seguro.

¿Desea habilitar el soporte de traducción con Angular Translate? Por defecto, JHipster proporciona un excelente soporte de internacionalización, tanto en el lado del cliente con Angular Translate como en el lado del servidor. Sin embargo, la internacionalización agrega un poco de sobrecarga y es un poco más complejo de administrar, por lo que puede elegir no instalar esta función.

Elige no si no estás seguro.

¿Qué marcos de prueba te gustaría usar? De forma predeterminada, JHipster proporciona pruebas de unidad / integración de Java (con el soporte JUnit de Spring) y pruebas de unidad de JavaScript (con Karma.js). Como opción, también puede agregar soporte para:

Elige ninguno si no estás seguro. Tendrás acceso a Junit y Karma por defecto.

Lea Aplicación web de arranque Spring-Responsive con JHipster en línea:

<https://riptutorial.com/es/spring-boot/topic/6297/aplicacion-web-de-arranque-spring-responsive-con-jhipster>

Capítulo 4: Bota de primavera + JPA + mongoDB

Examples

Operación CRUD en MongoDB usando JPA

Modelo de cliente

```
package org.bookmytickets.model;

import org.springframework.data.annotation.Id;

public class Customer {

    @Id
    private String id;
    private String firstName;
    private String lastName;

    public Customer() {}

    public Customer(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public Customer(String id, String firstName, String lastName) {
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
    }

    @Override
    public String toString() {
        return String.format(
            "Customer[id=%s, firstName='%s', lastName='%s']",
            id, firstName, lastName);
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
}
```

```

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}

```

Controlador del cliente

```

package org.bookmytickets.controller;

import java.util.List;

import org.bookmytickets.model.Customer;
import org.bookmytickets.repository.CustomerRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping(value = "/customer")
public class CustomerController {

    @Autowired
    private CustomerRepository repository;

    @GetMapping("")
    public List<Customer> selectAll(){
        List<Customer> customerList = repository.findAll();
        return customerList;
    }

    @GetMapping("/{id}")
    public List<Customer> getSpecificCustomer(@PathVariable String id){
        return repository.findById(id);
    }

    @GetMapping("/search/lastName/{lastName}")
    public List<Customer> searchByLastName(@PathVariable String lastName){
        return repository.findByLasttName(lastName);
    }

    @GetMapping("/search/firstName/{firstName}")
    public List<Customer> searchByFirstName(@PathVariable String firstName){
        return repository.findByFirstName(firstName);
    }

    @PostMapping("")
    public void insert(@RequestBody Customer customer) {
        repository.save(customer);
    }
}

```



```

@PatchMapping("/{id}")
public void update(@RequestParam String id, @RequestBody Customer customer) {
    Customer oldCustomer = repository.findById(id);
    if(customer.getFirstName() != null) {
        oldCustomer.setFristName(customer.getFirstName());
    }
    if(customer.getLastName() != null) {
        oldCustomer.setLastName(customer.getLastName());
    }
    repository.save(oldCustomer);
}

@DeleteMapping("/{id}")
public void delete(@RequestParam String id) {
    Customer deleteCustomer = repository.findById(id);
    repository.delete(deleteCustomer);
}
}

```

Repositorio de clientes

```

package org.bookmytickets.repository;

import java.util.List;

import org.bookmytickets.model.Customer;
import org.springframework.data.mongodb.repository.MongoRepository;

public interface CustomerRepository extends MongoRepository<Customer, String> {
    public Customer findByFirstName(String firstName);
    public List<Customer> findByLastName(String lastName);
}

```

pom.xml

Por favor agregue las dependencias abajo en el archivo pom.xml:

```

<dependencies>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-mongodb</artifactId>
    </dependency>

</dependencies>

```

Insertar datos usando rest client: método POST

Para probar nuestra aplicación, estoy usando el cliente de descanso avanzado, que es una extensión de Chrome:

Entonces, aquí está la instantánea para insertar los datos:

Request

> http://localhost:8080/customer/insert?firstName=Rakesh&lastName=Shankarnarayan

☐ GET

☒ POST

☐ PUT

☐ DELETE

Other methods

Custom content type

Raw headers

Headers form

Raw payload

Data form

Status: 200: OK ? Loading time: 112 ms

Obtener URL de solicitud

> http://localhost:8080/customer

☒ GET

☐ POST

☐ PUT

☐ DELETE

Other methods

Raw headers

Headers form

Obtener el resultado de la solicitud:

```
2]
-0: {
  "id": "579372b4a82615cd8b77af49"
  "firstName": "Raghu"
  "lastName": "Shankarnarayan"
}
-1: {
  "id": "5793b008a826191a3c5e9fcf"
  "firstName": "Rakesh"
  "lastName": "Shankarnarayan"
}
```

Lea Bota de primavera + JPA + mongoDB en línea: <https://riptutorial.com/es/spring-boot/topic/3398/bota-de-primavera-plus-jpa-plus-mongodb>

Capítulo 5: Bota de Primavera + JPA + RESTO

Observaciones

Este ejemplo utiliza Spring Boot, Spring Data JPA y Spring Data REST para exponer un simple objeto de dominio administrado por JPA a través de REST. El ejemplo responde con el formato HAL JSON y expone una url accesible en `/person`. La configuración de Maven incluye una base de datos en memoria H2 para admitir un soporte rápido.

Examples

Arranque de arranque de primavera

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {

    //main entry point
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

}
```

Objeto de dominio

```
package com.example.domain;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;

//simple person object with JPA annotations

@Entity
public class Person {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Long id;

    @Column
    private String firstName;

    @Column
```

```

private String lastName;

public Long getId() {
    return id;
}

public void setId(Long id) {
    this.id = id;
}

public String getFirstName() {
    return firstName;
}

public void setFirstName(String firstName) {
    this.firstName = firstName;
}

public String getLastName() {
    return lastName;
}

public void setLastName(String lastName) {
    this.lastName = lastName;
}
}

```

Interfaz de repositorio

```

package com.example.domain;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.repository.query.Param;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;
import org.springframework.data.rest.core.annotation.RestResource;

//annotation exposes the
@RepositoryRestResource(path="/person")
public interface PersonRepository extends JpaRepository<Person,Long> {

    //the method below allows us to expose a search query, customize the endpoint name, and
    specify a parameter name
    //the exposed URL is GET /person/search/byLastName?lastname=
    @RestResource(path="/byLastName")
    Iterable<Person> findByLastName(@Param("lastName") String lastName);

    //the methods below are examples on to prevent an operation from being exposed.
    //For example DELETE; the methods are overridden and then annotated with
    RestResource(exported=false) to make sure that no one can DELETE entities via REST
    @Override
    @RestResource(exported=false)
    default void delete(Long id) { }

    @Override
    @RestResource(exported=false)
    default void delete(Person entity) { }

    @Override

```

```

@RestResource(exported=false)
default void delete(Iterable<? extends Person> entities) { }

@Override
@RestResource(exported=false)
default void deleteAll() { }

@Override
@RestResource(exported=false)
default void deleteAllInBatch() { }

@Override
@RestResource(exported=false)
default void deleteInBatch(Iterable<Person> arg0) { }

}

```

Configuración de Maven

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.4.0.RELEASE</version>
  </parent>
  <groupId>com.example</groupId>
  <artifactId>spring-boot-data-jpa-rest</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>spring-boot-data-jpa-rest</name>
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-rest</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>

```

```
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
    </dependency>
</dependencies>
</project>
```

Lea Bota de Primavera + JPA + RESTO en línea: <https://riptutorial.com/es/spring-boot/topic/6507/bota-de-primavera-plus-jpa-plus-resto>

Capítulo 6: Conectando una aplicación de arranque de primavera a MySQL

Introducción

Sabemos que Spring-Boot se ejecuta de forma predeterminada utilizando la base de datos H2. En este artículo, veremos cómo modificar la configuración predeterminada para que funcione con la base de datos MySQL.

Observaciones

Como requisito previo, asegúrese de que MySQL ya se esté ejecutando en el puerto 3306 y de que se haya creado su base de datos.

Examples

Ejemplo de arranque de primavera usando MySQL

Seguiremos la [guía oficial para spring-boot y spring-data-jpa](#) . Vamos a construir la aplicación utilizando Gradle.

1. Crear el archivo de compilación de Gradle

construir.gradle

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:1.4.3.RELEASE")
    }
}

apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'idea'
apply plugin: 'org.springframework.boot'

jar {
    baseName = 'gs-accessing-data-jpa'
    version = '0.1.0'
}

repositories {
    mavenCentral()
    maven { url "https://repository.jboss.org/nexus/content/repositories/releases" }
}
```



```

sourceCompatibility = 1.8
targetCompatibility = 1.8

dependencies {
    compile("org.springframework.boot:spring-boot-starter-data-jpa")
    runtime('mysql:mysql-connector-java')
    testCompile("junit:junit")
}

```

2. Crea la entidad cliente

src / main / java / hello / Customer.java

```

@Entity
public class Customer {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Long id;
    private String firstName;
    private String lastName;

    protected Customer() {}

    public Customer(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    @Override
    public String toString() {
        return String.format(
            "Customer[id=%d, firstName='%s', lastName='%s']",
            id, firstName, lastName);
    }

}

```

3. Crea Repositorios

src / main / java / hello / CustomerRepository.java

```

import java.util.List;
import org.springframework.data.repository.CrudRepository;

public interface CustomerRepository extends CrudRepository<Customer, Long> {
    List<Customer> findByLastName(String lastName);
}

```

4. Crear el archivo application.properties

```

##### DataSource Configuration #####
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/your_database_name
jdbc.username=username
jdbc.password=password

```

```

init-db=false

##### Hibernate Configuration #####

hibernate.dialect=org.hibernate.dialect.MySQLDialect
hibernate.show_sql=true
hibernate.hbm2ddl.auto=update

```

5. Crea el archivo PersistenceConfig.java

En el paso 5, definiremos cómo se cargará la fuente de datos y cómo nuestra aplicación se conecta a MySQL. El fragmento de código anterior es la configuración mínima que necesitamos para conectarnos a MySQL. Aquí ofrecemos dos frijoles:

```

@Configuration
@EnableTransactionManagement
@EnableJpaRepositories(basePackages="hello")
public class PersistenceConfig
{
    @Autowired
    private Environment env;

    @Bean
    public LocalContainerEntityManagerFactoryBean entityManagerFactory()
    {
        LocalContainerEntityManagerFactoryBean factory = new
LocalContainerEntityManagerFactoryBean();

        HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
        vendorAdapter.setGenerateDdl(Boolean.TRUE);
        vendorAdapter.setShowSql(Boolean.TRUE);

        factory.setDataSource(dataSource());
        factory.setJpaVendorAdapter(vendorAdapter);
        factory.setPackagesToScan("hello");

        Properties jpaProperties = new Properties();
        jpaProperties.put("hibernate.hbm2ddl.auto",
env.getProperty("hibernate.hbm2ddl.auto"));
        factory.setJpaProperties(jpaProperties);

        factory.afterPropertiesSet();
        factory.setLoadTimeWeaver(new InstrumentationLoadTimeWeaver());
        return factory;
    }

    @Bean
    public DataSource dataSource()
    {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName(env.getProperty("jdbc.driverClassName"));
        dataSource.setUrl(env.getProperty("jdbc.url"));
        dataSource.setUsername(env.getProperty("jdbc.username"));
        dataSource.setPassword(env.getProperty("jdbc.password"));
        return dataSource;
    }
}

```

- **LocalContainerEntityManagerFactoryBean** Esto nos permite controlar las configuraciones de EntityManagerFactory y nos permite realizar personalizaciones. También nos permite inyectar el PersistenceContext en nuestros componentes de la siguiente manera:

```
@PersistenceContext
private EntityManager em;
```

- **Fuente de datos** Aquí devolvemos una instancia de DriverManagerDataSource . Es una implementación simple de la interfaz JDBC DataSource estándar, que configura un controlador JDBC antiguo y sencillo a través de las propiedades del bean, y devuelve una nueva conexión para cada llamada de getConnection. Tenga en cuenta que recomiendo usar esto estrictamente para propósitos de prueba, ya que hay mejores alternativas como BasicDataSource disponibles. Consulte [aquí](#) para más detalles.

6. Crear una clase de aplicación

src / main / java / hello / Application.java

```
@SpringBootApplication
public class Application {

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

    @Autowired
    private CustomerRepository repository;

    public static void main(String[] args) {
        SpringApplication.run(TestCoreApplication.class, args);
    }

    @Bean
    public CommandLineRunner demo() {
        return (args) -> {
            // save a couple of customers
            repository.save(new Customer("Jack", "Bauer"));
            repository.save(new Customer("Chloe", "O'Brian"));
            repository.save(new Customer("Kim", "Bauer"));
            repository.save(new Customer("David", "Palmer"));
            repository.save(new Customer("Michelle", "Dessler"));

            // fetch all customers
            log.info("Customers found with findAll():");
            log.info("-----");
            for (Customer customer : repository.findAll()) {
                log.info(customer.toString());
            }
            log.info("");

            // fetch an individual customer by ID
            Customer customer = repository.findOne(1L);
            log.info("Customer found with findOne(1L):");
            log.info("-----");
            log.info(customer.toString());
            log.info("");

            // fetch customers by last name
```

```

        log.info("Customer found with findByLastName('Bauer'):");
        log.info("-----");
        for (Customer bauer : repository.findByLastName("Bauer")) {
            log.info(bauer.toString());
        }
        log.info("");
    };
}
}
}

```

7. Ejecutando la aplicación

Si está utilizando un IDE como STS , puede simplemente hacer clic derecho en su proyecto -> Ejecutar como -> Generar Gradle (STS) ... En la lista de tareas, escriba bootRun y Ejecutar.

Si está utilizando gradle en la línea de comandos , simplemente puede ejecutar la aplicación de la siguiente manera:

```
./gradlew bootRun
```

Debería ver algo como esto:

```

== Customers found with findAll():
Customer[id=1, firstName='Jack', lastName='Bauer']
Customer[id=2, firstName='Chloe', lastName='O'Brian']
Customer[id=3, firstName='Kim', lastName='Bauer']
Customer[id=4, firstName='David', lastName='Palmer']
Customer[id=5, firstName='Michelle', lastName='Dessler']

== Customer found with findOne(1L):
Customer[id=1, firstName='Jack', lastName='Bauer']

== Customer found with findByLastName('Bauer'):
Customer[id=1, firstName='Jack', lastName='Bauer']
Customer[id=3, firstName='Kim', lastName='Bauer']

```

Lea Conectando una aplicación de arranque de primavera a MySQL en línea:

<https://riptutorial.com/es/spring-boot/topic/8588/conectando-una-aplicacion-de-arranque-de-primavera-a-mysql>

Capítulo 7: Controladores

Introducción

En esta sección agregaré un ejemplo para el controlador de resto de inicio de Spring con Get y post request.

Examples

Controlador de reposacabezas de muelle

En este ejemplo, mostraré cómo formular un controlador de descanso para obtener y publicar datos en la base de datos utilizando JPA con la mayor facilidad y el menor código.

En este ejemplo, nos referiremos a la tabla de datos llamada buyerRequirement.

BuyingRequirement.java

@Entity @Table (name = "BUYINGREQUIREMENTS") @NamedQueries ({@NamedQuery (name = "BuyingRequirement.findAll", query = "SELECT b FROM BuyingRequirement b"))) clase pública BuyingRequirement Extensiones de dominio Serializable {privado final final privado serialVersionUID = 1L;

```
@Column(name = "PRODUCT_NAME", nullable = false)
private String productname;

@Column(name = "NAME", nullable = false)
private String name;

@Column(name = "MOBILE", nullable = false)
private String mobile;

@Column(name = "EMAIL", nullable = false)
private String email;

@Column(name = "CITY")
private String city;

public BuyingRequirement() {
}

public String getProductname() {
    return productname;
}

public void setProductname(String productname) {
    this.productname = productname;
}
```

```

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public String getMobile() {
    return mobile;
}

public void setMobile(String mobile) {
    this.mobile = mobile;
}

public String getEmail() {
    return email;
}

public void setEmail(String email) {
    this.email = email;
}

public String getCity() {
    return city;
}

public void setCity(String city) {
    this.city = city;
}
}

```

Esta es la clase de entidad que incluye el parámetro que se refiere a las columnas en la tabla buyingRequirement y sus captadores y definidores.

IBuyingRequirementsRepository.java (interfaz JPA)

```

@Repository
@RepositoryRestResource
public interface IBuyingRequirementsRepository extends JpaRepository<BuyingRequirement, UUID>
{
    // Page<BuyingRequirement> findAllByOrderByCreatedDesc(Pageable pageable);
    Page<BuyingRequirement> findAllByOrderByCreatedDesc(Pageable pageable);
    Page<BuyingRequirement> findByNameContainingIgnoreCase(@Param("name") String name,
    Pageable pageable);
}

```

BuyingRequirementController.java

```

@RestController
@RequestMapping("/api/v1")
public class BuyingRequirementController {

    @Autowired
    IBuyingRequirementsRepository iBuyingRequirementsRepository;
}

```

```

Email email = new Email();

BuyerRequirementTemplate buyerRequirementTemplate = new BuyerRequirementTemplate();

private String To = "support@pharmerz.com";
// private String To = "amigujarathi@gmail.com";
private String Subject = "Buyer Request From Pharmerz ";

@PostMapping(value = "/buyingRequirement")
public ResponseEntity<BuyingRequirement> CreateBuyingRequirement (@RequestBody
BuyingRequirement buyingRequirements) {

    String productname = buyingRequirements.getProductname();
    String name = buyingRequirements.getName();
    String mobile = buyingRequirements.getMobile();
    String emails = buyingRequirements.getEmail();
    String city = buyingRequirements.getCity();
    if (city == null) {
        city = "-";
    }

    String HTMLBODY = buyerRequirementTemplate.template(productname, name, emails, mobile,
city);

    email.SendMail(To, Subject, HTMLBODY);

    iBuyingRequirementsRepository.save(buyingRequirements);
    return new ResponseEntity<BuyingRequirement>(buyingRequirements, HttpStatus.CREATED);
}

@GetMapping(value = "/buyingRequirements")
public Page<BuyingRequirement> getAllBuyingRequirements(Pageable pageable) {

    Page requirements =
iBuyingRequirementsRepository.findAllByOrderByCreatedDesc(pageable);
    return requirements;
}

@GetMapping(value = "/buyingRequirmentByName/{name}")
public Page<BuyingRequirement> getByName(@PathVariable String name,Pageable pageable){
    Page buyersByName =
iBuyingRequirementsRepository.findByNameContainingIgnoreCase(name,pageable);

    return buyersByName;
}
}

```

Incluye el método

1. Método de publicación que publica los datos en la base de datos.
2. Obtenga el método que obtiene todos los registros de la tabla de compras de requisitos.
3. Este es también un método de obtención que encontrará el requisito de compra por el nombre de la persona.

Lea Controladores en línea: <https://riptutorial.com/es/spring-boot/topic/10635/controladores>

Capítulo 8: Creación y uso de múltiples archivos de aplicaciones.propiedades

Examples

Dev y Prod entorno utilizando diferentes fuentes de datos

Después de configurar con éxito la aplicación Spring-Boot, toda la configuración se maneja en un archivo `application.properties`. Encontrará el archivo en `src/main/resources/`.

Normalmente es necesario tener una base de datos detrás de la aplicación. Para el desarrollo es bueno tener una configuración de `dev` y un entorno de `prod`. Mediante el uso de varios archivos `application.properties` puede indicar a Spring-Boot con qué entorno debe iniciar la aplicación.

Un buen ejemplo es configurar dos bases de datos. Uno para `dev` y otro para `productive`.

Para el `dev` entorno se puede utilizar una base de datos en memoria como `H2`. Cree un nuevo archivo en el directorio `src/main/resources/` llamado `application-dev.properties`. Dentro del archivo está la configuración de la base de datos en memoria:

```
spring.datasource.url=jdbc:h2:mem:test
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
```

Para el entorno `prod`, nos conectaremos a una base de datos "real", por ejemplo, `postgresql`. Cree un nuevo archivo en el directorio `src/main/resources/` llamado `application-prod.properties`. Dentro del archivo está la configuración de la base de datos `postgresql`:

```
spring.datasource.url= jdbc:postgresql://localhost:5432/yourDB
spring.datasource.username=postgres
spring.datasource.password=secret
```

En su archivo `application.properties` predeterminado `application.properties` ahora puede establecer qué perfil es activado y usado por Spring-Boot. Solo establece un atributo dentro:

```
spring.profiles.active=dev
```

o

```
spring.profiles.active=prod
```

Es importante que la parte posterior – en `application-dev.properties` sea el identificador del archivo.

Ahora puede iniciar la aplicación Spring-Boot en modo de desarrollo o producción simplemente

cambiando el identificador. Se iniciará una base de datos en memoria o la conexión a una base de datos "real". Claro que también hay muchos más casos de uso para tener múltiples archivos de propiedades.

Establezca el perfil de resorte correcto construyendo la aplicación automáticamente (maven)

Al crear múltiples archivos de propiedades para los diferentes entornos o casos de uso, a veces es difícil cambiar manualmente el valor de `active.profile` al correcto. Pero hay una manera de configurar el `active.profile` en el archivo `application.properties` mientras se construye la aplicación utilizando `maven-profiles`.

Digamos que hay tres entornos de archivos de propiedades en nuestra aplicación:

application-dev.properties :

```
spring.profiles.active=dev
server.port=8081
```

application-test.properties :

```
spring.profiles.active=test
server.port=8082
```

application-prod.properties .

```
spring.profiles.active=prod
server.port=8083
```

Esos tres archivos solo difieren en puerto y nombre de perfil activo.

En el archivo principal `application.properties` configuramos nuestro perfil de primavera usando una [variable maven](#) :

application.properties .

```
spring.profiles.active=@profileActive@
```

Después de eso solo tenemos que agregar los perfiles de Maven en nuestro `pom.xml` . Estableceremos los perfiles para los tres entornos:

```
<profiles>
  <profile>
    <id>dev</id>
    <activation>
      <activeByDefault>true</activeByDefault>
    </activation>
    <properties>
      <build.profile.id>dev</build.profile.id>
      <profileActive>dev</profileActive>
    </properties>
  </profile>
  <profile>
    <id>test</id>
    <activation>
      <activeByDefault>false</activeByDefault>
    </activation>
    <properties>
      <build.profile.id>test</build.profile.id>
      <profileActive>test</profileActive>
    </properties>
  </profile>
  <profile>
    <id>prod</id>
    <activation>
      <activeByDefault>false</activeByDefault>
    </activation>
    <properties>
      <build.profile.id>prod</build.profile.id>
      <profileActive>prod</profileActive>
    </properties>
  </profile>
</profiles>
```

```
</properties>
</profile>
<profile>
  <id>test</id>
  <properties>
    <build.profile.id>test</build.profile.id>
    <profileActive>test</profileActive>
  </properties>
</profile>
<profile>
  <id>prod</id>
  <properties>
    <build.profile.id>prod</build.profile.id>
    <profileActive>prod</profileActive>
  </properties>
</profile>
</profiles>
```

Ahora puedes construir la aplicación con maven. Si no configura ningún perfil de Maven, está creando el predeterminado (en este ejemplo es dev). Para especificar uno tienes que usar una palabra clave maven. La palabra clave para establecer un perfil en maven es `-P` seguida directamente por el nombre del perfil: `mvn clean install -Ptest`.

Ahora, también puede crear compilaciones personalizadas y guardarlas en su IDE para compilaciones más rápidas.

Ejemplos:

```
mvn clean install -Ptest
```

```

      .   _ _ _ _ _
/\ \ / _ _ ' _ _ _ _ ( _ ) _ _ _ _ \ \ \ \ \
( ( ) \ _ _ | ' _ | ' _ | | ' _ \ / _ ` | \ \ \ \ \
\ \ / _ _ ) | | _ | | | | | | | ( _ | | ) ) ) )
  ' | _ _ | . _ | _ | | _ | | _ \ _ , | / / / /
=====|_|=====| _ _ / = / _ / _ /
:: Spring Boot ::                (v1.5.3.RELEASE)

```

```
2017-06-06 11:24:44.885 INFO 6328 --- [           main] com.demo.SpringBlobApplicationTests
: Starting SpringApplicationTests on KB242 with PID 6328 (started by me in
C:\DATA\Workspaces\spring-demo)
2017-06-06 11:24:44.886 INFO 6328 --- [           main] com.demo.SpringApplicationTests
: The following profiles are active: test
```

```
mvn clean install -Pprod
```

```

      .      _ _ _ _ _
/\ \ / _ _ ' _ _ _ _ ( _ ) _ _ _ _ \ \ \ \ \
( ( ) \ _ _ | ' _ | ' _ | | ' _ \ / _ ` | \ \ \ \ \
\ \ / _ _ ) | | _ | | | | | | | ( _ | | ) ) ) )
      ' | _ _ | . _ | _ | | _ | _ \ _ , | / / / /
=====|_|=====| _ _ / = / _ / _ /
:: Spring Boot ::                (v1.5.3.RELEASE)

```

```
2017-06-06 14:43:31.067 INFO 6932 --- [           main] com.demo.SpringBlobApplicationTests
: Starting SpringApplicationTests on KB242 with PID 6328 (started by me in
C:\DATA\Workspaces\spring-demo)
```

```
2017-06-06 14:43:31.069 INFO 6932 --- [main] com.demo.SpringApplicationTests
: The following profiles are active: prod
```

Lea Creación y uso de múltiples archivos de aplicaciones.propiedades en línea:

<https://riptutorial.com/es/spring-boot/topic/6334/creacion-y-uso-de-multiples-archivos-de-aplicaciones-propiedades>

Capítulo 9: Escaneo de paquetes

Introducción

En este tema, haré una visión general de la exploración del paquete de arranque de primavera.

Puede encontrar información básica en Spring Boots en el siguiente enlace ([using-boot-structuring-your-code](#)) pero intentaré proporcionar información más detallada.

Spring boot, y spring en general, proporcionan una función para escanear automáticamente paquetes para ciertas anotaciones con el fin de crear `beans` y `configuration` .

Parámetros

Anotación	Detalles
@SpringBootApplication	Anotación principal de la aplicación de arranque de muelle. se utiliza una vez en la aplicación, contiene un método principal y actúa como paquete principal para el escaneo de paquetes
@SpringBootConfiguration	Indica que una clase proporciona la aplicación Spring Boot. Debe declararse solo una vez en la aplicación, generalmente de forma automática mediante la configuración de <code>@SpringBootApplication</code>
@EnableAutoConfiguration	Habilitar la configuración automática del contexto de la aplicación Spring. Debe declararse solo una vez en la aplicación, generalmente de forma automática mediante la configuración de <code>@SpringBootApplication</code>
@ComponentScan	Se utiliza para activar el escaneo automático de paquetes en un paquete determinado y sus hijos o para configurar el escaneo de paquetes personalizados
@Configuración	Se utiliza para declarar uno o más métodos <code>@Bean</code> . Puede seleccionarse mediante el escaneo automático de paquetes para declarar uno o más métodos <code>@Bean</code> lugar de la configuración xml tradicional
@Frijol	Indica que un método produce un bean para ser administrado por el contenedor Spring. Por <code>@Bean</code> general, los métodos anotados de <code>@Bean</code> se colocarán en las clases anotadas de <code>@Configuration</code> que se seleccionarán mediante el escaneo del paquete para crear beans basados en la configuración de Java.
@Componente	Al declarar una clase como <code>@Component</code> se convierte en un

Anotación	Detalles
	candidato para la detección automática cuando se utiliza la configuración basada en anotaciones y el escaneo de classpath. Por lo general, una clase anotada con <code>@Component</code> se convertirá en un <code>bean</code> en la aplicación
<code>@Repositorio</code>	Definido originalmente por Domain-Driven Design (Evans, 2003) como "un mecanismo para encapsular el almacenamiento. Normalmente se usa para indicar un <code>Repository</code> para <code>spring data</code>
<code>@Servicio</code>	Muy similar en la práctica a <code>@Component</code> . originalmente definido por Domain-Driven Design (Evans, 2003) como "una operación ofrecida como una interfaz que se mantiene aislada en el modelo, sin estado encapsulado".
<code>@Controlador</code>	Indica que una clase anotada es un "Controlador" (por ejemplo, un controlador web).
<code>@RestController</code>	Una anotación de conveniencia que se anota con <code>@Controller</code> y <code>@ResponseBody</code> . Se seleccionará automáticamente de forma predeterminada porque contiene la anotación <code>@Controller</code> que se selecciona de forma predeterminada.

Examples

@SpringBootApplication

La forma más básica de estructurar su código utilizando Spring Boot para un buen escaneo automático de paquetes es mediante la anotación `@SpringBootApplication` . Esta anotación proporciona en sí misma otras 3 anotaciones que ayudan con el escaneo automático:

`@SpringBootConfiguration` , `@EnableAutoConfiguration` , `@ComponentScan` (más información sobre cada anotación en la sección `Parameters`).

`@SpringBootApplication` lo `@SpringBootApplication` , `@SpringBootApplication` se colocará en el paquete principal y todos los demás componentes se colocarán en paquetes bajo este archivo:

```
com
+- example
  +- myproject
    +- Application.java (annotated with @SpringBootApplication)
    |
    +- domain
    |   +- Customer.java
    |   +- CustomerRepository.java
    |
    +- service
    |   +- CustomerService.java
    |
```

```
+-- web
    +- CustomerController.java
```

A menos que se indique lo contrario, Spring Boot detecta las `@Configuration` , `@Component` , `@Repository` , `@Service` , `@Controller` , `@RestController` automáticamente en los paquetes escaneados (`@Configuration` y `@RestController` se seleccionan porque están anotadas por `@Component` y `@Controller` consecuencia).

Ejemplo de código básico:

```
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

}
```

Configurando paquetes / clases explícitamente

Desde la versión **1.3** también puede decirle a Spring Boot que `scanBasePackages` paquetes específicos configurando `scanBasePackages` o `scanBasePackageClasses` en `@SpringBootApplication` en lugar de especificar `@ComponentScan` .

1. `@SpringBootApplication(scanBasePackages = "com.example.myproject")` : establece `com.example.myproject` como el paquete base para escanear.
2. `@SpringBootApplication(scanBasePackageClasses = CustomerController.class)` : una alternativa segura para el `scanBasePackages` a `scanBasePackages` establece el paquete de `CustomerController.java` , `com.example.myproject.web` , como el paquete base para escanear.

Excluyendo la autoconfiguración

Otra característica importante es la capacidad de excluir clases específicas de configuración automática utilizando `exclude` o `excludeName` (existe `excludeName` desde la versión **1.3**).

1. `@SpringBootApplication(exclude = DemoConfiguration.class)` : excluirá `DemoConfiguration` de la exploración automática de paquetes.
2. `@SpringBootApplication(excludeName = "DemoConfiguration")` - hará lo mismo usando el nombre completamente clasificado de la clase.

@ComponentScan

Puede usar `@ComponentScan` para configurar un análisis de paquetes más complejo. También hay `@ComponentScans` que actúan como una anotación de contenedor que agrega varias anotaciones de `@ComponentScan` .

Ejemplos de código básico

```
@ComponentScan
```

```
public class DemoAutoConfiguration {  
}
```

```
@ComponentScans({@ComponentScan("com.example1"), @ComponentScan("com.example2")})  
public class DemoAutoConfiguration {  
}
```

La indicación de `@ComponentScan` sin configuración actúa como `@SpringBootApplication` y analiza todos los paquetes de la clase anotada con esta anotación.

En este ejemplo, `@ComponentScan` algunos de los atributos útiles de `@ComponentScan` :

1. **Los paquetes base** se pueden usar para indicar paquetes específicos para escanear.
2. **useDefaultFilters** : al establecer este atributo en falso (el valor predeterminado es verdadero), puede asegurarse de que Spring no escanee `@Component` , `@Repository` , `@Service` o `@Controller` automáticamente.
3. **includeFilters** : se puede usar para *incluir* anotaciones de primavera / patrones de expresiones regulares para incluir en el escaneo de paquetes.
4. **excludeFilters** : se pueden usar para *excluir* anotaciones específicas de primavera / patrones de expresiones regulares para incluir en el escaneo de paquetes.

Hay muchos más atributos, pero esos son los más utilizados para personalizar el escaneo de paquetes.

Creando tu propia autoconfiguración.

Spring Boot se basa en muchos proyectos primarios de autoconfiguración prefabricados. Ya debe estar familiarizado con los proyectos de arranque de arranque de primavera.

Puede crear fácilmente su propio proyecto de inicio siguiendo estos sencillos pasos:

1. Cree algunas clases de `@Configuration` para definir beans predeterminados. Debe utilizar propiedades externas tanto como sea posible para permitir la personalización y tratar de utilizar las anotaciones del ayudante de configuración automática como `@AutoConfigureBefore` , `@AutoConfigureAfter` , `@ConditionalOnBean` , `@ConditionalOnMissingBean` etc. Se puede encontrar información más detallada sobre cada anotación en los oficiales documentación [anotaciones Estado](#)
2. Coloque un archivo / archivos de configuración automática que agregue todas las clases de `@Configuration` .
3. Cree un archivo llamado `spring.factories` y colóquelo en `src/main/resources/META-INF` .
4. En `spring.factories` , establezca la propiedad `org.springframework.boot.autoconfigure.EnableAutoConfiguration` con valores separados por comas de sus clases de `@Configuration` :

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\ncom.mycorp.libx.autoconfigure.LibXAutoConfiguration,\ncom.mycorp.libx.autoconfigure.LibXWebAutoConfiguration
```

Usando este método, puede crear sus propias clases de configuración automática que serán seleccionadas por spring-boot. Spring-boot escanea automáticamente todas las dependencias de maven / gradle en busca de un archivo `spring.factories` , si encuentra uno, agrega todas `@Configuration` clases de `@Configuration` especificadas en él a su proceso de configuración automática.

Asegúrese de que su proyecto de inicio de `auto-configuration` no contenga `spring boot maven plugin` ya que empaquetará el proyecto como un JAR ejecutable y no será cargado por el classpath como se `spring.factories` no cargará tu configuración

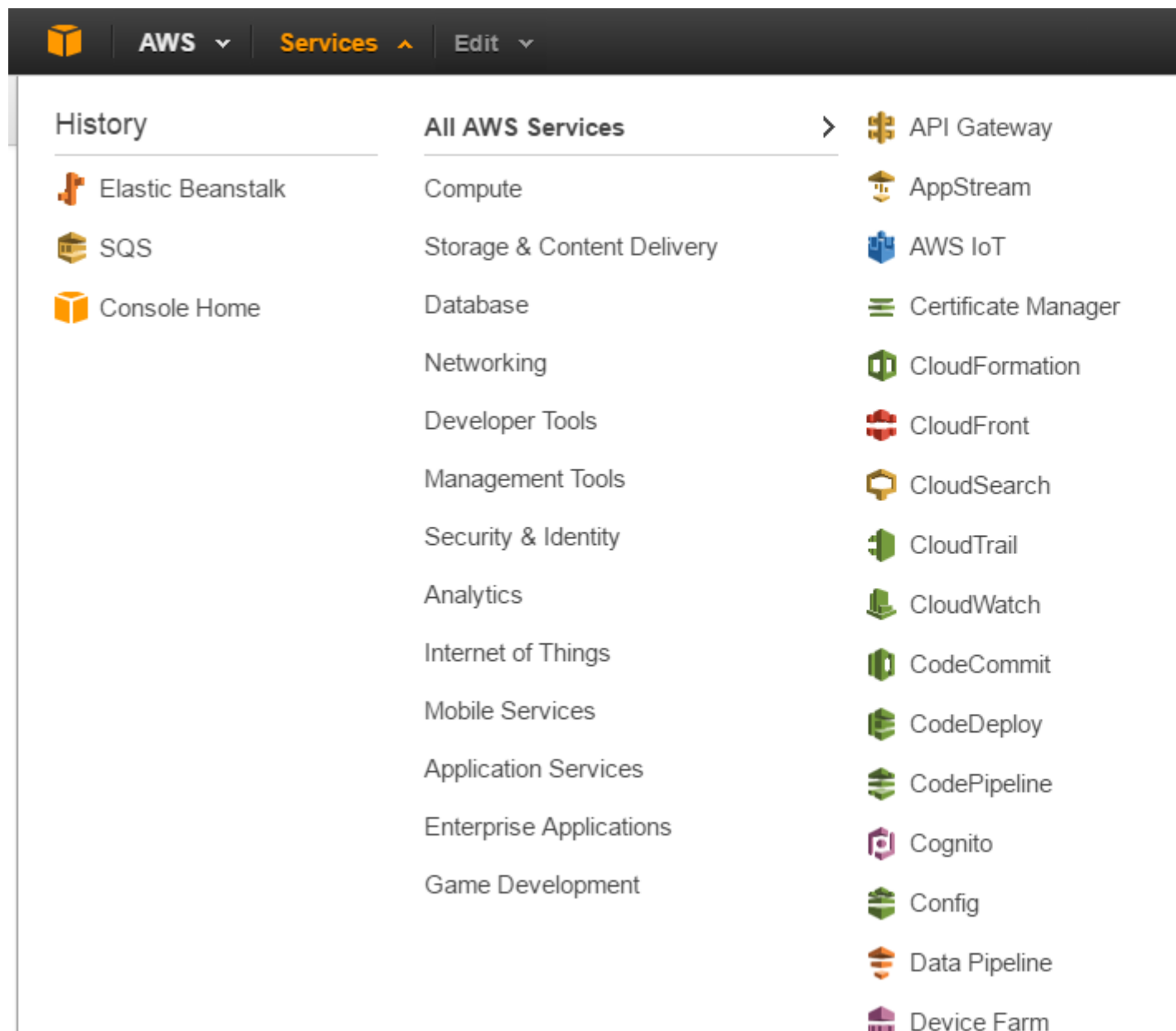
Lea Escaneo de paquetes en línea: <https://riptutorial.com/es/spring-boot/topic/9354/escaneo-de-paquetes>

Capítulo 10: Implementación de la aplicación de ejemplo utilizando Spring-boot en Amazon Elastic Beanstalk


Examples


Implementación de una aplicación de ejemplo utilizando Spring-boot en formato Jar en AWS


1. Cree una aplicación de ejemplo utilizando spring-boot desde [el](#) sitio de [inicializador](#) de [spring-boot](#) .
2. Importe el código en su IDE local y ejecute el objetivo como **una instalación limpia spring-boot: run -e**
3. Vaya a la carpeta de destino y busque el archivo jar.
4. Abra su cuenta de Amazon o cree una nueva cuenta de Amazon y seleccione para Elastic Beanstalk como se muestra a continuación




5. Cree un nuevo entorno de servidor web como se muestra a continuación

AWS

Services

Edit

Elastic Beanstalk

My First Elastic Beanstalk Application

New Environment

Environment Type

Application Version

Environment Info

Additional Resources

Configuration Details

Environment Tags

Permissions

Review Information

New Environment


AWS Elastic Beanstalk has two types of environment tiers to support different process HTTP requests, typically over port 80. Workers are specialized applications that process messages in a queue. Worker applications post those messages to your application by using the Amazon SQS API.

Web Server Environment


Provides resources for an AWS Elastic Beanstalk web server in either a single instance or an auto scaling environment. [Learn more.](#)

Worker Environment*

Provides resources for an AWS Elastic Beanstalk worker application in either a single instance or an auto scaling environment. [Learn more.](#)

 * Worker environments require additional permissions to access Amazon SQS.

6. Seleccione el tipo de entorno como Java para la implementación del archivo **JAR** para Spring-boot, si planea implementarlo como un archivo **WAR** , debe seleccionarse como tomcat como se muestra a continuación

AWS

Services

Edit

Elastic Beanstalk

My First Elastic Beanstalk Application

New Environment

Environment Type

Application Version

Environment Info

Additional Resources

Configuration Details

Environment Tags

Permissions

Review Information


Environment Type

Choose the platform and type of environment to launch.

Predefined configuration: Java ▼ Loc...

AWS Elastic Beanstalk will create an environment

Environment type: Load balancing, auto scaling ▼ Lea...

AWS

Services

Edit

Elastic Beanstalk

My First Elastic Beanstalk Application

New Environment

Environment Type

Application Version

Environment Info

Additional Resources

Configuration Details

Environment Tags

Permissions

Review Information

Environment Type

Choose the platform and type of environment to launch.

Predefined configuration: Tomcat ▼ L...

AWS Elastic Beanstalk will create an environ

Environment type: Load balancing, auto scaling ▼ L...

7. Seleccione con la configuración predeterminada al hacer clic en siguiente siguiente ...
8. Una vez que complete la configuración predeterminada, en la pantalla de resumen, el archivo JAR se puede cargar e implementar como se muestra en las figuras.

[All Applications](#) > [My First Elastic Beanstalk Application](#) > [Default-Env](#)

[est-2.elasticbeanstalk.com](#))

Dashboard

Configuration

Logs

Health

Monitoring

Alarms

Managed Updates **NEW**

Events

Tags

Overview



Health

Ok

Causes

Recent Events

Time	Type	Details
2016-08-27 03:36:06 UTC+0530	INFO	Environment health has
2016-08-27 03:35:06 UTC+0530	WARN	Environment health has
2016-08-26 13:15:58 UTC+0530	INFO	Deleted log fragments f
2016-08-26 13:12:47 UTC+0530	INFO	Environment health has

Upload and Deploy

i To deploy a previous version, go to the Application Versions page.


Upload application: No file chosen


Version label:

► Deployment Preferences

Current number of instances: 1

9. Una vez que la Implementación sea exitosa (5 -10 minutos por primera vez), puede golpear la url de contexto como se muestra en la siguiente figura.

 **AWS** ▾ **Services** ▾ **Edit** ▾


 Elastic Beanstalk **My First Elastic Beanstalk Application** ▾

[All Applications](#) > [My First Elastic Beanstalk Application](#) > [Default-Env](#)
([est-2.elasticbeanstalk.com](#))

Dashboard Overview

Configuration

Logs

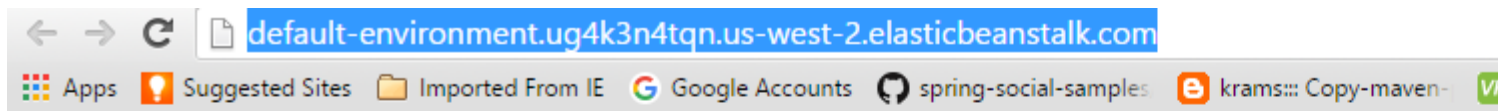
Health  **Health**
Ok

Monitoring

Alarms

Managed Updates **NEW** Recent Events

10. El resultado es el que se muestra a continuación, debería funcionar igual que con su env. Local.



Demo Boot

11. Por favor encuentra mi [URL de Github](#)

Lea Implementación de la aplicación de ejemplo utilizando Spring-boot en Amazon Elastic Beanstalk en línea: <https://riptutorial.com/es/spring-boot/topic/6117/implementacion-de-la-aplicacion-de-ejemplo-utilizando-spring-boot-en-amazon-elastic-beanstalk>

Capítulo 11: Instalación de la CLI de inicio de Spring

Introducción

La [CLI de Spring Boot](#) le permite crear y trabajar fácilmente con las aplicaciones de Spring Boot desde la línea de comandos.

Observaciones

Una vez instalado, el CLI de inicio de Spring se puede ejecutar con el comando `spring` :

Para obtener ayuda de línea de comandos:

```
$ spring help
```

Para crear y ejecutar su primer proyecto de arranque de primavera:

```
$ spring init my-app
$ cd my-app
$ spring run my-app
```

Abra su navegador para `localhost:8080` :

```
$ open http://localhost:8080
```

Obtendrá la página de error de whitelabel porque aún no ha agregado ningún recurso a su aplicación, pero está listo para ir con los siguientes archivos:

```
my-app/
├─ mvnw
├─ mvnw.cmd
├─ pom.xml
├─ src/
│   └─ main/
│       ├── java/
│       │   └─ com/
│       │       └─ example/
│       │           └─ DemoApplication.java
│       └─ resources/
│           └─ application.properties
└─ test/
    └─ java/
        └─ com/
            └─ example/
                └─ DemoApplicationTests.java
```

- `mvnw` y `mvnw.cmd` : scripts de envoltorio de Maven que descargarán e instalarán Maven (si es

necesario) en el primer uso.

- `pom.xml` - La definición del proyecto Maven
- `DemoApplication.java` : la clase principal que inicia su aplicación Spring Boot.
- `application.properties` : un archivo para las propiedades de configuración externalizadas. (También se puede dar una extensión `.yml`.)
- `DemoApplicationTests.java` : una prueba unitaria que valida la inicialización del contexto de la aplicación Spring Boot.

Examples

Instalación manual

Consulte la [página de descarga](#) para descargar y descomprimir manualmente la última versión, o siga los enlaces a continuación:

- [spring-boot-cli-1.5.1.RELEASE-bin.zip](#)
- [spring-boot-cli-1.5.1.RELEASE-bin.tar.gz](#)

Instalar en Mac OSX con HomeBrew

```
$ brew tap pivotal/tap
$ brew install springboot
```

Instalar en Mac OSX con MacPorts

```
$ sudo port install spring-boot-cli
```

Instalar en cualquier sistema operativo con SDKMAN!

SDKMAN! es el administrador de kit de desarrollo de software para Java. Se puede usar para instalar y administrar versiones de la CLI de Spring Boot, así como Java, Maven, Gradle y más.

```
$ sdk install springboot
```

Lea Instalación de la CLI de inicio de Spring en línea: <https://riptutorial.com/es/spring-boot/topic/9031/instalacion-de-la-cli-de-inicio-de-spring>

Capítulo 12: Microservicio de arranque de primavera con JPA

Examples

Clase de aplicación

```
package com.mcf7.spring;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringDataMicroServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringDataMicroServiceApplication.class, args);
    }
}
```

Modelo de libro

```
package com.mcf7.spring.domain;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
import java.io.Serializable;

@lombok.Getter
@lombok.Setter
@lombok.EqualsAndHashCode(of = "isbn")
@lombok.ToString(exclude="id")
@Entity
public class Book implements Serializable {

    public Book() {}

    @Id
    @GeneratedValue(strategy= GenerationType.AUTO)
    private long id;

    @NotNull
    @Size(min = 1)
    private String isbn;

    @NotNull
    @Size(min = 1)
    private String title;
}
```

```

@NotNull
@Size(min = 1)
private String author;

@NotNull
@Size(min = 1)
private String description;
}

```

Solo una nota ya que algunas cosas están sucediendo aquí, quería romperlas muy rápido.

Todas las anotaciones con `@lombok` están generando algunas de las placas de calderas de nuestra clase.

```

@lombok.Getter //Creates getter methods for our variables

@lombok.Setter //Creates setter methods four our variables

@lombok.EqualsAndHashCode(of = "isbn") //Creates Equals and Hashcode methods based off of the
isbn variable

@lombok.ToString(exclude="id") //Creates a toString method based off of every variable except
id

```

También aprovechamos la Validación en este Objeto.

```

@NotNull //This specifies that when validation is called this element shouldn't be null

@Size(min = 1) //This specifies that when validation is called this String shouldn't be
smaller than 1

```

Repositorio de libros

```

package com.mcf7.spring.domain;

import org.springframework.data.repository.PagingAndSortingRepository;

public interface BookRepository extends PagingAndSortingRepository<Book, Long> {
}

```

Patrón básico de Spring Repository, excepto que habilitamos un repositorio de búsqueda y clasificación para funciones adicionales como ... paginación y clasificación :)

Habilitar la validación

```

package com.mcf7.spring.domain;

import org.springframework.validation.Errors;
import org.springframework.validation.Validator;

public class BeforeCreateBookValidator implements Validator{
    public boolean supports(Class<?> clazz) {

```

```

        return Book.class.equals(clazz);
    }

    public void validate(Object target, Errors errors) {
        errors.reject("rejected");
    }
}

```

Cargando algunos datos de prueba

```

package com.mcf7.spring.domain;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

@Component
public class DatabaseLoader implements CommandLineRunner {
    private final BookRepository repository;

    @Autowired
    public DatabaseLoader(BookRepository repository) {
        this.repository = repository;
    }

    public void run(String... Strings) throws Exception {
        Book book1 = new Book();
        book1.setIsbn("6515616168418510");
        book1.setTitle("SuperAwesomeTitle");
        book1.setAuthor("MCF7");
        book1.setDescription("This Book is super epic!");
        repository.save(book1);
    }
}

```

Simplemente cargando algunos datos de prueba, idealmente, esto debería agregarse solo en un perfil de desarrollo.

Añadiendo el validador

```

package com.mcf7.spring.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;
import org.springframework.data.rest.core.event.ValidatingRepositoryEventListener;
import org.springframework.data.rest.webmvc.config.RepositoryRestConfigurerAdapter;
import org.springframework.validation.Validator;
import org.springframework.validation.beanvalidation.LocalValidatorFactoryBean;

@Configuration
public class RestValidationConfiguration extends RepositoryRestConfigurerAdapter {

    @Bean
    @Primary
    /**
     * Create a validator to use in bean validation - primary to be able to autowire without

```

```

qualifier
    */
    Validator validator() {
        return new LocalValidatorFactoryBean();
    }

    @Override
    public void configureValidatingRepositoryEventListener(ValidatingRepositoryEventListener
validatingListener) {
        Validator validator = validator();
        //bean validation always before save and create
        validatingListener.addValidator("beforeCreate", validator);
        validatingListener.addValidator("beforeSave", validator);
    }
}

```

Gradle Build File

```

buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'io.spring.gradle:dependency-management-plugin:0.5.4.RELEASE'
    }
}

apply plugin: 'io.spring.dependency-management'
apply plugin: 'idea'
apply plugin: 'java'

dependencyManagement {
    imports {
        mavenBom 'io.spring.platform:platform-bom:2.0.5.RELEASE'
    }
}

sourceCompatibility = 1.8
targetCompatibility = 1.8

repositories {
    mavenCentral()
}

dependencies {
    compile 'org.springframework.boot:spring-boot-starter-web'
    compile 'org.springframework.boot:spring-boot-starter-data-jpa'
    compile 'org.springframework.boot:spring-boot-starter-data-rest'
    compile 'org.springframework.data:spring-data-rest-hal-browser'
    compile 'org.projectlombok:lombok:1.16.6'
    compile 'org.springframework.boot:spring-boot-starter-validation'
    compile 'org.springframework.boot:spring-boot-actuator'

    runtime 'com.h2database:h2'

    testCompile 'org.springframework.boot:spring-boot-starter-test'
    testCompile 'org.springframework.restdocs:spring-restdocs-mockmvc'
}

```

Lea Microservicio de arranque de primavera con JPA en línea: <https://riptutorial.com/es/spring-boot/topic/6557/microservicio-de-arranque-de-primavera-con-jpa>

Capítulo 13: Pruebas en Spring Boot

Examples

Cómo probar una aplicación de arranque de primavera simple

Tenemos una aplicación de arranque Spring de muestra que almacena los datos del usuario en MongoDB y estamos usando los servicios Rest para recuperar datos.

Primero hay una clase de dominio, es decir, POJO

```
@Document
public class User{
    @Id
    private String id;

    private String name;
}
```

Un repositorio correspondiente basado en Spring Data MongoDB

```
public interface UserRepository extends MongoRepository<User, String> {
}
```

Entonces nuestro controlador de usuario

```
@RestController
class UserController {

    @Autowired
    private UserRepository repository;

    @RequestMapping("/users")
    List<User> users() {
        return repository.findAll();
    }

    @RequestMapping(value = "/Users/{id}", method = RequestMethod.DELETE)
    @ResponseStatus(HttpStatus.NO_CONTENT)
    void delete(@PathVariable("id") String id) {
        repository.delete(id);
    }

    // more controller methods
}
```

Y finalmente nuestra aplicación de arranque de primavera.

```
@SpringBootApplication
public class Application {
    public static void main(String args[]){
```

```

    SpringApplication.run(Application.class, args);
}
}

```

Si, digamos que John Cena, The Rock y TripleHHH eran los únicos tres usuarios en la base de datos, una solicitud a / users daría la siguiente respuesta:

```

$ curl localhost:8080/users
[{"name":"John Cena","id":"1"}, {"name":"The Rock","id":"2"}, {"name":"TripleHHH","id":"3"}]

```

Ahora para probar el código verificaremos que la aplicación funcione.

```

@RunWith(SpringJUnit4ClassRunner.class)    // 1
@SpringApplicationConfiguration(classes = Application.class)    // 2
@WebAppConfiguration    // 3
@IntegrationTest("server.port:0")    // 4
public class UserControllerTest {

    @Autowired    // 5
    UserRepository repository;

    User cena;
    User rock;
    User tripleHHH;

    @Value("${local.server.port}")    // 6
    int port;

    @Before
    public void setUp() {
        // 7
        cena = new User("John Cena");
        rock = new User("The Rock");
        tripleHHH = new User("TripleHH");

        // 8
        repository.deleteAll();
        repository.save(Arrays.asList(cena, rock, tripleHHH));

        // 9
        RestAssured.port = port;
    }

    // 10
    @Test
    public void testFetchCena() {
        String cenaId = cena.getId();

        when().
            get("/Users/{id}", cenaId).
        then().
            statusCode(HttpStatus.SC_OK).
            body("name", Matchers.is("John Cena")).
            body("id", Matchers.is(cenaId));
    }

    @Test
    public void testFetchAll() {

```



```

        when().
            get("/users").
        then().
            statusCode(HttpStatus.SC_OK).
            body("name", Matchers.hasItems("John Cena", "The Rock", "TripleHHH"));
    }

    @Test
    public void testDeletetripleHHH() {
        String tripleHHHId = tripleHHH.getId();

        when().
            delete("/Users/{id}", tripleHHHId).
        then().
            statusCode(HttpStatus.SC_NO_CONTENT);
    }
}

```

Explicación

1. Como cualquier otra prueba basada en Spring, necesitamos el `SpringJUnit4ClassRunner` para que se `SpringJUnit4ClassRunner` un contexto de aplicación.
2. La anotación `@SpringApplicationConfiguration` es similar a la anotación `@ContextConfiguration` en que se usa para especificar qué contexto (s) de aplicación deben usarse en la prueba. Además, activará la lógica para leer las configuraciones específicas de Spring Boot, las propiedades, etc.
3. `@WebAppConfiguration` debe estar presente para indicar a Spring que se debe cargar un `WebApplicationContext` para la prueba. También proporciona un atributo para especificar la ruta a la raíz de la aplicación web.
4. `@IntegrationTest` se utiliza para indicar a Spring Boot que se debe iniciar el servidor web incorporado. Al proporcionar pares de nombre-valor separados por dos puntos o iguales, cualquier variable de entorno puede ser anulada. En este ejemplo, `"server.port:0"` anulará la configuración de puerto predeterminada del servidor. Normalmente, el servidor comenzaría a usar el número de puerto especificado, pero el valor 0 tiene un significado especial. Cuando se especifica como 0, le dice a Spring Boot que escanee los puertos en el entorno host e inicie el servidor en un puerto disponible aleatorio. Esto es útil si tenemos diferentes servicios que ocupan diferentes puertos en las máquinas de desarrollo y el servidor de compilación que podría colisionar con el puerto de la aplicación, en cuyo caso la aplicación no se iniciará. En segundo lugar, si creamos múltiples pruebas de integración con diferentes contextos de aplicación, también pueden chocar si las pruebas se ejecutan simultáneamente.
5. Tenemos acceso al contexto de la aplicación y podemos usar el cableado automático para inyectar cualquier Spring Bean.
6. El valor de `@Value("${local.server.port}")` se resolverá con el número de puerto real que se utiliza.
7. Creamos algunas entidades que podemos utilizar para la validación.
8. La base de datos MongoDB se borra y reinicializa para cada prueba, de modo que siempre se validen en un estado conocido. Dado que el orden de las pruebas no está definido, es probable que la prueba `testFetchAll()` falle si se ejecuta después de la prueba `testDeletetripleHHH()`.

9. Le [indicamos](#) a [Rest Assured](#) que use el puerto correcto. Es un proyecto de código abierto que proporciona un DSL de Java para probar servicios de descanso.
10. Las pruebas se implementan utilizando Rest Assured. podemos implementar las pruebas utilizando TestRestTemplate o cualquier otro cliente http, pero uso Rest Assured porque podemos escribir documentación concisa utilizando [RestDocs](#)

Cargando diferentes archivos yaml [o propiedades] o anular algunas propiedades

Cuando usamos `@SpringApplicationConfiguration`, usará la configuración de `application.yml` [propiedades] que en ciertas situaciones no es apropiada. Entonces, para anular las propiedades podemos usar la anotación `@TestPropertySource`.

```
@TestPropertySource(  
    properties = {  
        "spring.jpa.hibernate.ddl-auto=create-drop",  
        "liquibase.enabled=false"  
    }  
)  
@RunWith(SpringJUnit4ClassRunner.class)  
@SpringApplicationConfiguration(Application.class)  
public class ApplicationTest{  
  
    // ...  
  
}
```

Podemos usar el atributo de **propiedades** de `@TestPropertySource` para anular las **propiedades** específicas que deseamos. En el ejemplo anterior, estamos **anulando la propiedad** `spring.jpa.hibernate.ddl-auto` a `create-drop`. Y `liquibase.enabled` a `false`.

Cargando diferentes archivos yaml

Si desea cargar totalmente un archivo **yaml** diferente para la prueba, puede usar el atributo de **ubicaciones** en `@TestPropertySource`.

```
@TestPropertySource(locations="classpath:test.yml")  
@RunWith(SpringJUnit4ClassRunner.class)  
@SpringApplicationConfiguration(Application.class)  
public class ApplicationTest{  
  
    // ...  
  
}
```

Opciones alternativas

Opción 1:

También puede cargar **un** archivo **yml** diferente **si** coloca un archivo **yml** en la `test > resource` directorio de `test > resource`

Opcion 2:

Usando la anotación `@ActiveProfiles`

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = Application.class)
@ActiveProfiles("somename")
public class MyIntTest{
}
```

Puede ver que estamos usando la anotación `@ActiveProfiles` y estamos pasando el **nombre** como el valor.

Cree un archivo llamado `application-somename.yml` y la prueba cargará este archivo.

Lea Pruebas en Spring Boot en línea: <https://riptutorial.com/es/spring-boot/topic/1985/pruebas-en-spring-boot>

Capítulo 14: Servicios de descanso

Parámetros

Anotación	Columna
@Controlador	Indica que una clase anotada es un "Controlador" (controlador web).
@RequestMapping	Anotación para mapear solicitudes web en clases de manejadores específicos (si usamos con clase) y / o métodos de manejador (si usamos con métodos).
method = RequestMethod.GET	Tipo de métodos de solicitud HTTP
Cuerpo de Respuesta	La anotación que indica un valor de retorno del método debe estar vinculada al cuerpo de respuesta web
@RestController	@Controller +.ResponseBody
@ResponseBody	Extensión de HttpEntity que agrega un código de estado HttpStatus, podemos controlar el código http de retorno

Examples

Creación de un servicio REST

1. Cree un proyecto utilizando STS (Spring Starter Project) o Spring Initializr (en <https://start.spring.io>).
2. Agregue una dependencia web en su pom.xml:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

o escriba **web** en el cuadro de búsqueda `Search for dependencies` , agregue la dependencia web y descargue el proyecto comprimido.

3. Crear una clase de dominio (es decir, usuario)

```
public class User {

    private Long id;
```

```

private String userName;

private String password;

private String email;

private String firstName;

private String lastName;

public Long getId() {
    return id;
}

public void setId(Long id) {
    this.id = id;
}

public String getUser_name() {
    return userName;
}

public void setUser_name(String userName) {
    this.userName = userName;
}

public String getPassword() {
    return password;
}

public void setPassword(String password) {
    this.password = password;
}

public String getEmail() {
    return email;
}

public void setEmail(String email) {
    this.email = email;
}

public String getFirstName() {
    return firstName;
}

public void setFirstName(String firstName) {
    this.firstName = firstName;
}

public String getLastName() {
    return lastName;
}

public void setLastName(String lastName) {
    this.lastName = lastName;
}

@Override
public String toString() {
    return "User [id=" + id + ", userName=" + userName + ", password=" + password + ",

```

```

email=" + email
        + ", firstName=" + firstName + ", lastName=" + lastName + "]";
    }

    public User(Long id, String userName, String password, String email, String firstName,
String lastName) {
        super();
        this.id = id;
        this.userName = userName;
        this.password = password;
        this.email = email;
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public User() {}
}

```

4. Cree la clase UserController y agregue @Controller , @RequestMapping anotaciones

```

@Controller
@RequestMapping(value = "api")
public class UserController {
}

```

5. Defina la variable de usuarios de la lista estática para simular la base de datos y agregar 2 usuarios a la lista

```

private static List<User> users = new ArrayList<User>();

public UserController() {
    User u1 = new User(1L, "shijazi", "password", "shijazi88@gmail.com", "Safwan",
"Hijazi");
    User u2 = new User(2L, "test", "password", "test@gmail.com", "test", "test");
    users.add(u1);
    users.add(u2);
}

```

6. Crear un nuevo método para devolver a todos los usuarios en la lista estática (getAllUsers)

```

@RequestMapping(value = "users", method = RequestMethod.GET)
public @ResponseBody List<User> getAllUsers() {
    return users;
}

```

7. Ejecute la aplicación [por mvn clean install spring-boot:run] y llame a esta URL

<http://localhost:8080/api/users>

8. Podemos anotar la clase con @RestController , y en este caso podemos eliminar el **ResponseBody** de todos los métodos de esta clase, (@RestController = @Controller + ResponseBody) , un punto más podemos controlar el código http de retorno si usamos **ResponseEntity** , implementaremos las mismas funciones anteriores pero utilizando **@RestController y ResponseEntity**

```

@RestController
@RequestMapping(value = "api2")
public class UserController2 {

    private static List<User> users = new ArrayList<User>();

    public UserController2() {
        User u1 = new User(1L, "shijazi", "password", "shijazi88@gmail.com", "Safwan",
"Hijazi");
        User u2 = new User(2L, "test", "password", "test@gmail.com", "test", "test");
        users.add(u1);
        users.add(u2);
    }

    @RequestMapping(value = "users", method = RequestMethod.GET)
    public ResponseEntity<?> getAllUsers() {
        try {
            return new ResponseEntity<>(users, HttpStatus.OK);
        } catch (Exception e) {
            return new ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR);
        }
    }
}

```

ahora intente ejecutar la aplicación y llame a esta URL [http: // localhost: 8080 / api2 / users](http://localhost:8080/api2/users)

Creando un servicio de descanso con JERSEY y Spring Boot

Jersey es uno de los muchos marcos disponibles para crear servicios de descanso. Este ejemplo le mostrará cómo crear servicios de descanso con Jersey y Spring Boot.

1. Configuración del proyecto

Puede crear un nuevo proyecto utilizando STS o utilizando la página [Spring Initializr](#) . Mientras creas un proyecto, incluye las siguientes dependencias:

1. Jersey (JAX-RS)
2. Web

2. Crear un controlador

Permítanos crear un controlador para nuestro servicio web de Jersey.

```

@Path("/Welcome")
@Component
public class MyController {
    @GET
    public String welcomeUser(@QueryParam("user") String user){
        return "Welcome "+user;
    }
}

```

`@Path("/Welcome")` indica al marco que este controlador debe responder a la ruta URI / Welcome

`@QueryParam("user")` indica al marco que estamos esperando un parámetro de consulta con el nombre `user`

Configuraciones de Jersey de cableado

Ahora configuremos Jersey Framework con Spring Boot:

`org.glassfish.jersey.server.ResourceConfig` una clase, en lugar de un componente spring que se extiende `org.glassfish.jersey.server.ResourceConfig` :

```
@Component
@ApplicationPath("/MyRestService")
public class JerseyConfig extends ResourceConfig {
    /**
     * Register all the Controller classes in this method
     * to be available for jersey framework
     */
    public JerseyConfig() {
        register(MyController.class);
    }
}
```

`@ApplicationPath("/MyRestService")` indica al marco que solo las solicitudes dirigidas a la ruta /MyRestService deben ser manejadas por el marco de Jersey, otras solicitudes deben continuar siendo manejadas por el marco de Spring.

Es una buena idea anotar la clase de configuración con `@ApplicationPath` , de lo contrario, todas las solicitudes serán manejadas por Jersey y no podremos omitirlas y dejar que un controlador de resorte lo maneje si es necesario.

4. Hecho

Inicie la aplicación y active una URL de muestra como (asumiendo que ha configurado Spring Boot para ejecutarse en el puerto 8080):

`http://localhost:8080/MyRestService/Welcome?user=User`

Debería ver un mensaje en su navegador como:

Bienvenido Usuario

Y ya ha terminado con su servicio web de Jersey con Spring Boot.

Consumir una API REST con RestTemplate (GET)

Para consumir una API REST con `RestTemplate` , cree un proyecto de inicio Spring con el inicio bootzr Spring y asegúrese de que se agrega la dependencia **Web** :


```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Una vez [que haya configurado su proyecto](#) , cree un bean `RestTemplate` . Puede hacer esto dentro de la clase principal que ya se ha generado, o dentro de una clase de configuración separada (una clase anotada con `@Configuration`):

```
@Bean
public RestTemplate restTemplate() {
    return new RestTemplate();
}
```

Después de eso, cree una clase de dominio, similar a lo que debe hacer al [crear un servicio REST](#) .

```
public class User {
    private Long id;
    private String username;
    private String firstname;
    private String lastname;

    public Long getId() {
        return id;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getFirstname() {
        return firstname;
    }

    public void setFirstname(String firstname) {
        this.firstname = firstname;
    }

    public String getLastname() {
        return lastname;
    }

    public void setLastname(String lastname) {
        this.lastname = lastname;
    }
}
```

En su cliente, autowire el `RestTemplate` :

```
@Autowired
private RestTemplate restTemplate;
```

Para consumir una API REST que devuelve un solo usuario, ahora puede usar:

```
String url = "http://example.org/path/to/api";
User response = restTemplate.getForObject(url, User.class);
```

Al consumir una API REST que devuelve una lista o una matriz de usuarios, tiene dos opciones. O bien consumirlo como una matriz:

```
String url = "http://example.org/path/to/api";
User[] response = restTemplate.getForObject(url, User[].class);
```

O consumirlo usando la `ParameterizedTypeReference` :

```
String url = "http://example.org/path/to/api";
ResponseEntity<List<User>> response = restTemplate.exchange(url, HttpMethod.GET, null, new
ParameterizedTypeReference<List<User>>() {});
List<User> data = response.getBody();
```

Tenga en cuenta que, cuando use `ParameterizedTypeReference` , tendrá que usar el método más avanzado `RestTemplate.exchange()` y tendrá que crear una subclase. En el ejemplo anterior, se utiliza una clase anónima.

Lea Servicios de descanso en línea: <https://riptutorial.com/es/spring-boot/topic/1920/servicios-de-descanso>

Capítulo 15: Spring boot + Hibernate + Web UI (Thymeleaf)

Introducción

Este hilo se enfoca en cómo crear una aplicación de arranque de resorte con hibernación y motor de plantilla de hoja de timón.

Observaciones

También [puedes](#) ver la [documentación de Thymeleaf](#)

Examples

Dependencias maven

Este ejemplo se basa en Spring Boot 1.5.1.RELEASE. Con las siguientes dependencias:

```
<!-- Spring -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<!-- Lombok -->
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <optional>true</optional>
</dependency>
<!-- H2 -->
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
</dependency>
<!-- Test -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
```

En este ejemplo vamos a utilizar Spring Boot JPA, Thymeleaf y web starters. Estoy usando

Lombok para generar getters y setters más fáciles pero no es obligatorio. H2 se utilizará como una base de datos en la memoria fácil de configurar.

Configuración de hibernación

Primero, veamos lo que necesitamos para configurar Hibernate correctamente.

1. `@EnableTransactionManagement` y `@EnableJpaRepositories` : queremos una gestión transaccional y utilizar repositorios de datos Spring.
2. `DataSource` - fuente de datos principal para la aplicación. utilizando en memoria h2 para este ejemplo.
3. `LocalContainerEntityManagerFactoryBean` : fábrica de gestores de entidades de primavera que utiliza `HibernateJpaVendorAdapter` .
4. `PlatformTransactionManager` - gestor de transacciones principal para `@Transactional` anotó componentes.

Archivo de configuración:

```
@Configuration
@EnableTransactionManagement
@EnableJpaRepositories(basePackages = "com.example.repositories")
public class PersistenceJpaConfig {

    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("org.h2.Driver");
        dataSource.setUrl("jdbc:h2:mem:testdb;mode=MySQL;DB_CLOSE_DELAY=-1;DB_CLOSE_ON_EXIT=FALSE");
        dataSource.setUsername("sa");
        dataSource.setPassword("");
        return dataSource;
    }

    @Bean
    public LocalContainerEntityManagerFactoryBean entityManagerFactory(DataSource dataSource)
    {
        LocalContainerEntityManagerFactoryBean em = new
LocalContainerEntityManagerFactoryBean();
        em.setDataSource(dataSource);
        em.setPackagesToScan(new String[] { "com.example.models" });
        JpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
        em.setJpaVendorAdapter(vendorAdapter);
        em.setJpaProperties(additionalProperties());
        return em;
    }

    @Bean
    public PlatformTransactionManager
transactionManager(LocalContainerEntityManagerFactoryBean entityManagerFactory, DataSource
dataSource) {
        JpaTransactionManager tm = new JpaTransactionManager();
        tm.setEntityManagerFactory(entityManagerFactory.getObject());
        tm.setDataSource(dataSource);
        return tm;
    }
}
```

```

    Properties additionalProperties() {
        Properties properties = new Properties();
        properties.setProperty("hibernate.hbm2ddl.auto", "update");
        properties.setProperty("hibernate.dialect", "org.hibernate.dialect.H2Dialect");
        return properties;
    }
}

```

Entidades y Repositorios

Una entidad simple: utilizar las `@Getter` Lombok `@Getter` y `@Setter` para generar `@Getter` y `@Setter` para nosotros

```

@Entity
@Getter @Setter
public class Message {

    @Id
    @GeneratedValue(generator = "system-uuid")
    @GenericGenerator(name = "system-uuid", strategy = "uuid")
    private String id;
    private String message;
}

```

Estoy usando IDs y lombok basados en UUID para generar captadores y definidores.

Un repositorio simple para la entidad anterior:

```

@Transactional
public interface MessageRepository extends CrudRepository<Message, String> {
}

```

Más sobre respositories: [documentos de datos de primavera](#)

Asegúrese de que las entidades residen en un paquete que está asignado en el `em.setPackagesToScan` (definido en el frijol `LocalContainerEntityManagerFactoryBean`) y los repositorios en un paquete asignado en los `basePackages` (definidos en la anotación `@EnableJpaRepositories`)

Recursos Thymeleaf y Spring Controller

Para exponer las plantillas de Thymeleaf necesitamos definir controladores.

Ejemplo:

```

@Controller
@RequestMapping("/")
public class MessageController {

    @Autowired
    private MessageRepository messageRepository;
}

```

```

@GetMapping
public ModelAndView index() {
    Iterable<Message> messages = messageRepository.findAll();
    return new ModelAndView("index", "index", messages);
}
}

```

Este controlador simple inyecta `MessageRepository` y pasa todos los mensajes a un archivo de plantilla llamado `index.html` , que reside en `src/main/resources/templates` , y finalmente lo expone en `/index` .

De la misma manera, podemos colocar otras plantillas en la carpeta de plantillas (por defecto, Spring to `src/main/resources/templates`), pasarles un modelo y entregarlas al cliente.

Otros recursos estáticos se deben colocar en una de las siguientes carpetas, expuestas de forma predeterminada en Spring Boot:

```

/META-INF/resources/
/resources/
/static/
/public/

```

Ejemplo de Thymeleaf `index.html` :

```

<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
  <head th:fragment="head (title)">
    <title th:text="${title}">Index</title>
    <link rel="stylesheet" th:href="@{/css/bootstrap.min.css}"
href="../../css/bootstrap.min.css" />
  </head>
  <body>
    <nav class="navbar navbar-default navbar-fixed-top">
      <div class="container-fluid">
        <div class="navbar-header">
          <a class="navbar-brand" href="#">Thymeleaf</a>
        </div>
      </div>
    </nav>
    <div class="container">
      <ul class="nav">
        <li><a th:href="@{/}" href="messages.html"> Messages </a></li>
      </ul>
    </div>
  </body>
</html>

```

- `bootstrap.min.css` está en la carpeta `src/main/resources/static/css` . puede usar la sintaxis `@{ }` para obtener otros recursos estáticos usando la ruta relativa.

Lea Spring boot + Hibernate + Web UI (Thymeleaf) en línea: <https://riptutorial.com/es/spring-boot/topic/9200/spring-boot-plus-hibernate-plus-web-ui--thymeleaf>

Capítulo 16: Spring Boot + Spring Data Elasticsearch

Introducción

[Spring Data Elasticsearch](#) es una implementación de [Spring Data](#) para [Elasticsearch](#) que proporciona integración con el motor de búsqueda [Elasticsearch](#).

Examples

Integración de Spring Boot y Spring Data Elasticsearch

En este ejemplo, vamos a implementar el proyecto spring-data-elasticsearch para almacenar POJO en elasticsearch. Veremos un proyecto de maven muestra que hace lo siguiente:

- Inserte un elemento de `Greeting(id, username, message)` en elasticsearch.
- Obtener todos los elementos de saludo que se han insertado.
- Actualizar un elemento de saludo.
- Eliminar un elemento de saludo.
- Obtener un elemento de saludo por id.
- Obtener todos los artículos de saludo por nombre de usuario.

Integración elasticsearch de arranque y datos de primavera.

En este ejemplo, vamos a ver una aplicación de arranque de primavera basada en Maven que integra Spring-Data-elasticsearch. Aquí, haremos lo siguiente y veremos los segmentos de código respectivos.

- Inserte un elemento de `Greeting(id, username, message)` en elasticsearch.
- Obtener todos los artículos de elasticsearch
- Actualizar un elemento específico.
- Eliminar un elemento específico.
- Obtener un elemento específico por id.
- Obtener un artículo específico por nombre de usuario.

Archivo de configuración del proyecto (pom.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.springdataes</groupId>
```

```

<artifactId>springdataes</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>jar</packaging>

<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.5.6.RELEASE</version>
</parent>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-elasticsearch</artifactId>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <executions>
        <execution>
          <goals>
            <goal>repackage</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
</project>

```

Usaremos [Spring Boot](#) de la versión 1.5.6.RELEASE y [Spring Data Elasticsearch](#) de esa versión respectiva. Para este proyecto, necesitamos ejecutar [elasticsearch-2.4.5](#) para probar nuestros apis.

Archivo de propiedades

Pondremos el archivo de propiedades del proyecto (llamado `applications.properties`) en `resources` carpeta de `resources` que contiene:

```

elasticsearch.clustername = elasticsearch
elasticsearch.host = localhost
elasticsearch.port = 9300

```

Usaremos el nombre de clúster, el host y el puerto predeterminados. De forma predeterminada, el puerto 9300 se utiliza como puerto de transporte y el puerto 9200 se conoce como puerto http. Para ver el nombre del clúster predeterminado, pulse [http: // localhost: 9200 /](http://localhost:9200/) .

Clase principal (Application.java)


```

package org.springdataes;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {
    public static void main(String []args) {
        SpringApplication.run(Application.class, args);
    }
}

```

`@SpringBootApplication` es una combinación de `@Configuration`, `@EnableAutoConfiguration`, `@EnableWebMvc` y `@ComponentScan` anotaciones. El método `main()` utiliza el método `SpringApplication.run()` Spring Boot para iniciar una aplicación. No necesitamos ninguna configuración xml, esta aplicación es una aplicación java spring pura.

Clase de configuración de Elasticsearch (ElasticsearchConfig.java)

```

package org.springdataes.config;

import org.elasticsearch.client.Client;
import org.elasticsearch.client.transport.TransportClient;
import org.elasticsearch.common.settings.Settings;
import org.elasticsearch.common.transport.InetSocketTransportAddress;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;
import org.springframework.data.elasticsearch.core.ElasticsearchOperations;
import org.springframework.data.elasticsearch.core.ElasticsearchTemplate;
import org.springframework.data.elasticsearch.repository.config.EnableElasticsearchRepositories;
import java.net.InetAddress;

@Configuration
@PropertySource(value = "classpath:applications.properties")
@EnableElasticsearchRepositories(basePackages = "org.springdataes.dao")
public class ElasticsearchConfig {
    @Value("${elasticsearch.host}")
    private String EsHost;

    @Value("${elasticsearch.port}")
    private int EsPort;

    @Value("${elasticsearch.clustername}")
    private String EsClusterName;

    @Bean
    public Client client() throws Exception {
        Settings esSettings = Settings.settingsBuilder()
            .put("cluster.name", EsClusterName)
            .build();

        return TransportClient.builder()
            .settings(esSettings)
            .build()
            .addTransportAddress(new

```

```

InetSocketAddress(InetAddress.getByName(EsHost), EsPort));
    }

    @Bean
    public ElasticsearchOperations elasticsearchTemplate() throws Exception {
        return new ElasticsearchTemplate(client());
    }
}

```

ElasticsearchConfig **clase** ElasticsearchConfig configura elasticsearch para este proyecto y establece una conexión con elasticsearch. Aquí, se utiliza `@PropertySource` para leer el archivo `application.properties` donde almacenamos el nombre del clúster, el host de elasticsearch y el puerto. `@EnableElasticsearchRepositories` se utiliza para habilitar los repositorios de Elasticsearch que escanearán los paquetes de la clase de configuración anotada para los repositorios de Spring Data de forma predeterminada. `@Value` se usa aquí para leer las propiedades del archivo `application.properties`.

El método `client()` crea una conexión de transporte con elasticsearch. La configuración anterior configura un servidor de Elasticsearch integrado que es utilizado por `ElasticsearchTemplate`. El bean `ElasticsearchTemplate` utiliza el `Elasticsearch Client` y proporciona una capa personalizada para manipular datos en Elasticsearch.

Clase modelo (Greeting.java)

```

package org.springdataes.model;

import org.springframework.data.annotation.Id;
import org.springframework.data.elasticsearch.annotations.Document;
import java.io.Serializable;

@Document(indexName = "index", type = "greetings")
public class Greeting implements Serializable{

    @Id
    private String id;

    private String username;

    private String message;

    public Greeting() {
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
    }
}

```

```

        this.username = username;
    }

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}

```

Aquí hemos anotado nuestros objetos de datos de `Greeting` con una anotación `@Document` que también podemos utilizar para determinar la configuración del índice como el nombre, el número de fragmentos o el número de réplicas. Uno de los atributos de la clase debe ser un `id`, ya sea `@Id` con `@Id` o usando uno de los nombres que se encuentran automáticamente como `id` o `documentId`. Aquí, el valor del campo `id` generará automáticamente, si no configuramos ningún valor del campo `id`.

Clase de repositorio de Elasticsearch (GreetingRepository.class)

```

package org.springdataes.dao;

import org.springdataes.model.Greeting;
import org.springframework.data.elasticsearch.repository.ElasticsearchRepository;
import java.util.List;

public interface GreetingRepository extends ElasticsearchRepository<Greeting, String> {
    List<Greeting> findByUsername(String username);
}

```

Aquí, hemos ampliado `ElasticsearchRepository` que nos proporciona muchas de las API que no necesitamos definir externamente. Esta es la clase de repositorio base para las clases de dominio basadas en `elasticsearch`. Dado que amplía las clases de repositorio basadas en `Spring`, obtenemos el beneficio de evitar el código de repetición requerido para implementar capas de acceso a datos para varios almacenes de persistencia.

Aquí hemos declarado un método `findByUsername(String username)` que se convertirá en una consulta de coincidencia que coincide con el nombre de usuario con el campo de `username` de `Objetos de Greeting` y devuelve la lista de resultados.

Servicios (GreetingService.java)

```

package org.springdataes.service;

import org.springdataes.model.Greeting;
import java.util.List;

public interface GreetingService {
    List<Greeting> getAll();
    Greeting findOne(String id);
    Greeting create(Greeting greeting);
    Greeting update(Greeting greeting);
    List<Greeting> getGreetingByUsername(String username);
}

```

```
void delete(String id);  
}
```

Servicio Bean (GreetingServiceBean.java)

```
package org.springdataes.service;  
  
import com.google.common.collect.Lists;  
import org.springdataes.dao.GreetingRepository;  
import org.springdataes.model.Greeting;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
  
import java.util.List;  
  
@Service  
public class GreetingServiceBean implements GreetingService {  
  
    @Autowired  
    private GreetingRepository repository;  
  
    @Override  
    public List<Greeting> getAll() {  
        return Lists.newArrayList(repository.findAll());  
    }  
  
    @Override  
    public Greeting findOne(String id) {  
        return repository.findOne(id);  
    }  
  
    @Override  
    public Greeting create(Greeting greeting) {  
        return repository.save(greeting);  
    }  
  
    @Override  
    public Greeting update(Greeting greeting) {  
        Greeting persistedGreeting = repository.findOne(greeting.getId());  
        if(persistedGreeting == null) {  
            return null;  
        }  
        return repository.save(greeting);  
    }  
  
    @Override  
    public List<Greeting> getGreetingByUsername(String username) {  
        return repository.findByUsername(username);  
    }  
  
    @Override  
    public void delete(String id) {  
        repository.delete(id);  
    }  
}
```

En la clase anterior, hemos `@Autowired` the `GreetingRepository` . Simplemente podemos llamar a los métodos de `CRUDRepository` y al método que hemos declarado en la clase de repositorio con el objeto `GreetingRepository` .

En el método `getAll()` , puede encontrar una línea `Lists.newArrayList(repository.findAll())` . Hemos hecho esto para convertir `repository.findAll()` en el elemento `List<>` , ya que devuelve una lista `Iterable` .

Clase de controlador (GreetingController.java)

```
package org.springdataes.controller;

import org.springdataes.model.Greeting;
import org.springdataes.service.GreetingService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api")
public class GreetingController {

    @Autowired
    private GreetingService greetingService;

    @ResponseBody
    @RequestMapping(value = "/greetings", method = RequestMethod.GET)
    public ResponseEntity<List<Greeting>> getAll() {
        return new ResponseEntity<List<Greeting>>(greetingService.getAll(), HttpStatus.OK);
    }

    @ResponseBody
    @RequestMapping(value = "/greetings", method = RequestMethod.POST)
    public ResponseEntity<Greeting> insertGreeting(@RequestBody Greeting greeting) {
        return new ResponseEntity<Greeting>(greetingService.create(greeting),
        HttpStatus.CREATED);
    }

    @ResponseBody
    @RequestMapping(value = "/greetings", method = RequestMethod.PUT)
    public ResponseEntity<Greeting> updateGreeting(@RequestBody Greeting greeting) {
        return new ResponseEntity<Greeting>(greetingService.update(greeting),
        HttpStatus.MOVED_PERMANENTLY);
    }

    @ResponseBody
    @RequestMapping(value = "/greetings/{id}", method = RequestMethod.DELETE)
    public ResponseEntity<Greeting> deleteGreeting(@PathVariable("id") String id) {
        greetingService.delete(id);
        return new ResponseEntity<Greeting>(HttpStatus.NO_CONTENT);
    }

    @ResponseBody
    @RequestMapping(value = "/greetings/{id}", method = RequestMethod.POST)
    public ResponseEntity<Greeting> getOne(@PathVariable("id") String id) {
        return new ResponseEntity<Greeting>(greetingService.findOne(id), HttpStatus.OK);
    }

    @ResponseBody
    @RequestMapping(value = "/greetings/{name}", method = RequestMethod.GET)
    public ResponseEntity<List<Greeting>> getByUserName(@PathVariable("name") String name) {
```

```
        return new ResponseEntity<List<Greeting>>(greetingService.getGreetingByUsername(name),
        HttpStatus.OK);
    }
}
```

Construir

Para construir esta aplicación maven ejecutar

```
mvn clean install
```

El comando anterior primero elimina todos los archivos en la carpeta de `target` y crea el proyecto. Después de construir el proyecto obtendremos el archivo ejecutable `.jar` que se llama `springdataes-1.0-SNAPSHOT.jar`. Podemos ejecutar la clase principal (`Application.java`) para iniciar el proceso o simplemente ejecutando el tarro anterior escribiendo:

```
java -jar springdataes-1.0-SNAPSHOT.jar
```

Comprobando las APIs

Para insertar un elemento de saludo en elasticsearch, ejecute el siguiente comando

```
curl -H "Content-Type: application/json" -X POST -d '{"username":"sunkuet02","message": "this is a message"}' http://localhost:8080/api/greetings
```

Debería obtener el siguiente resultado como

```
{"id":"AV2ddRxBcuirs1TrVgHH","username":"sunkuet02","message":"this is a message"}
```

También puede verificar la API de obtención ejecutando:

```
curl -H "Content-Type: application/json" -X GET http://localhost:8080/api/greetings
```

Usted debe obtener

```
[{"id":"AV2ddRxBcuirs1TrVgHH","username":"sunkuet02","message":"this is a message"}]
```

Puedes consultar otras apis siguiendo los procesos anteriores.

Documentaciones oficiales:

- <https://projects.spring.io/spring-boot/>
- <https://projects.spring.io/spring-data-elasticsearch/>

Lea Spring Boot + Spring Data Elasticsearch en línea: <https://riptutorial.com/es/spring-boot/topic/10928/spring-boot-plus-spring-data-elasticsearch>

Capítulo 17: Spring boot + Spring Data JPA

Introducción

Spring Boot facilita la creación de aplicaciones y servicios impulsados por Spring, grado de producción con un mínimo esfuerzo absoluto. Favorece la convención sobre la configuración.

Spring Data JPA, parte de la gran familia de **Spring Data**, facilita la implementación de repositorios basados en JPA. Facilita la creación de aplicaciones que utilizan tecnologías de acceso a datos.

Observaciones

Anotaciones

`@Repository`: indica que una clase anotada es un "Repository", un mecanismo para encapsular el comportamiento de almacenamiento, recuperación y búsqueda que emula una colección de objetos. Los equipos que implementan patrones J2EE tradicionales como "Objeto de acceso a datos" también pueden aplicar este estereotipo a las clases DAO, aunque se debe tener cuidado para comprender la distinción entre Objetos de acceso a datos y repositorios de estilo DDD antes de hacerlo. Esta anotación es un estereotipo de propósito general y los equipos individuales pueden restringir su semántica y usar según sea apropiado.

`@RestController`: una anotación de conveniencia que se anota con `@Controller` y `@ResponseBody.Types` `@Controller` que llevan esta anotación se tratan como controladores donde los métodos de `@RequestMapping` asumen la semántica de `@ResponseBody` de manera predeterminada.

`@Service`: indica que una clase anotada es un "Servicio" (por ejemplo, una fachada de servicios empresariales). Esta anotación sirve como una especialización de `@Component`, lo que permite que las clases de implementación se `@Component` automáticamente a través del escaneo de classpath.

`@SpringBootApplication`: muchos desarrolladores de Spring Boot siempre tienen su clase principal anotada con `@Configuration`, `@EnableAutoConfiguration` y `@ComponentScan`. Dado que estas anotaciones se usan juntas con tanta frecuencia (especialmente si sigue las mejores prácticas mencionadas anteriormente), Spring Boot ofrece una alternativa conveniente a `@SpringBootApplication`.

`@Entity`: especifica que la clase es una entidad. Esta anotación se aplica a la clase de entidad.

Documentacion oficial

Pivotal Software ha proporcionado una documentación bastante extensa sobre Spring Framework, y se puede encontrar en

- <https://projects.spring.io/spring-boot/>
- <http://projects.spring.io/spring-data-jpa/>
- <https://spring.io/guides/gs/accessing-data-jpa/>

Examples

Ejemplo básico de integración de Spring Boot y Spring Data JPA

Vamos a construir una aplicación que almacene POJOs en una base de datos. La aplicación utiliza Spring Data JPA para almacenar y recuperar datos en una base de datos relacional. Su característica más atractiva es la capacidad de crear implementaciones de repositorio automáticamente, en tiempo de ejecución, desde una interfaz de repositorio.

Clase principal

```
package org.springframework.boot;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

El método `main()` utiliza el método `SpringApplication.run()` Spring Boot para iniciar una aplicación. Tenga en cuenta que no hay una sola línea de XML. Ningún archivo `web.xml` tampoco. Esta aplicación web es 100% pura de Java y no tiene que lidiar con la configuración de tuberías o infraestructura.

Clase de entidad

```
package org.springframework.model;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Greeting {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String text;

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



```

    }

    public Greeting(String text) {
        super();
        this.text = text;
    }

    /* In this example, the typical getters and setters have been left out for brevity. */
}

```

Aquí tienes una clase de `Greeting` con dos atributos, el `id` y el `text`. También tienes dos constructores. El constructor predeterminado solo existe por el bien de JPA. No lo usarás directamente, por lo que puede ser designado como `protected`. El otro constructor es el que usarás para crear instancias de `Greeting` que se guardarán en la base de datos.

La clase de `Greeting` está anotada con `@Entity`, lo que indica que es una entidad JPA. Por falta de una anotación de `@Table`, se supone que esta entidad se asignará a una tabla llamada "Saludo".

La propiedad `id` del saludo se anota con `@Id` para que JPA la reconozca como el ID del objeto. La propiedad `id` también se anota con `@GeneratedValue` para indicar que la ID debe generarse automáticamente.

La otra propiedad, el `text` se deja sin anotar. Se supone que se asignará a una columna que comparte el mismo nombre que la propiedad en sí.

Propiedades transitorias

En una clase de entidad similar a la anterior, podemos tener propiedades que no queremos que se conserven en la base de datos o se creen como columnas en nuestra base de datos, tal vez porque solo queremos establecerlas en el tiempo de ejecución y usarlas en nuestra aplicación. por lo tanto, podemos tener esa propiedad anotada con la anotación `@Transient`.

```

package org.springframework.model;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Transient;

@Entity
public class Greeting {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String text;
    @Transient
    private String textInSomeLanguage;

    public Greeting() {
        super();
    }
}

```

```

public Greeting(String text) {
    super();
    this.text = text;
    this.textInSomeLanguage = getTextTranslationInSpecifiedLanguage(text);
}

/* In this example, the typical getters and setters have been left out for brevity. */
}

```

Aquí tiene la misma clase de saludo que ahora tiene una propiedad transitoria `textInSomeLanguage` que puede inicializarse y usarse en tiempo de ejecución y no se conservará en la base de datos.

Clase DAO

```

package org.springframework.repository;

import org.springframework.model.Greeting;
import org.springframework.data.repository.CrudRepository;

public interface GreetingRepository extends CrudRepository<Greeting, Long> {

    List<Greeting> findByText(String text);
}

```

`GreetingRepository` extiende la interfaz de `CrudRepository`. El tipo de entidad y la ID con la que trabaja, `Greeting` y `Long`, se especifican en los parámetros genéricos en `CrudRepository`. Al extender `CrudRepository`, `GreetingRepository` hereda varios métodos para trabajar con la persistencia del `Greeting`, incluidos los métodos para guardar, eliminar y encontrar entidades de `Greeting`.

Consulte [esta discusión](#) para comparar `CrudRepository`, `PagingAndSortingRepository`, `JpaRepository`.

Spring Data JPA también le permite definir otros métodos de consulta simplemente declarando su firma de método. En el caso de `GreetingRepository`, esto se muestra con un método `findByText()`.

En una aplicación Java típica, esperarías escribir una clase que implemente `GreetingRepository`. Pero eso es lo que hace que Spring Data JPA sea tan poderoso: no tiene que escribir una implementación de la interfaz del repositorio. Spring Data JPA crea una implementación sobre la marcha cuando ejecuta la aplicación.

Clase de servicio

```

package org.springframework.service;

import java.util.Collection;
import org.springframework.model.Greeting;

public interface GreetingService {

    Collection<Greeting> findAll();
    Greeting findOne(Long id);
}

```

```
Greeting create(Greeting greeting);
Greeting update(Greeting greeting);
void delete(Long id);

}
```

Servicio de frijol

```
package org.springframework.service;

import java.util.Collection;
import org.springframework.model.Greeting;
import org.springframework.repository.GreetingRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class GreetingServiceBean implements GreetingService {

    @Autowired
    private GreetingRepository greetingRepository;

    @Override
    public Collection<Greeting> findAll() {
        Collection<Greeting> greetings = greetingRepository.findAll();
        return greetings;
    }

    @Override
    public Greeting findOne(Long id) {
        Greeting greeting = greetingRepository.findOne(id);
        return greeting;
    }

    @Override
    public Greeting create(Greeting greeting) {
        if (greeting.getId() != null) {
            //cannot create Greeting with specified Id value
            return null;
        }
        Greeting savedGreeting = greetingRepository.save(greeting);
        return savedGreeting;
    }

    @Override
    public Greeting update(Greeting greeting) {
        Greeting greetingPersisted = findOne(greeting.getId());
        if (greetingPersisted == null) {
            //cannot find Greeting with specified Id value
            return null;
        }
        Greeting updatedGreeting = greetingRepository.save(greeting);
        return updatedGreeting;
    }

    @Override
    public void delete(Long id) {
        greetingRepository.delete(id);
    }
}
```

```
}
```

Clase de controlador

```
package org.springframework.web.api;

import java.util.Collection;
import org.springframework.model.Greeting;
import org.springframework.service.GreetingService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping(value = "/api")
public class GreetingController {

    @Autowired
    private GreetingService greetingService;

    // GET [method = RequestMethod.GET] is a default method for any request.
    // So we do not need to mention explicitly

    @RequestMapping(value = "/greetings", produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<Collection<Greeting>> getGreetings() {
        Collection<Greeting> greetings = greetingService.findAll();
        return new ResponseEntity<Collection<Greeting>>(greetings, HttpStatus.OK);
    }

    @RequestMapping(value = "/greetings/{id}", produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<Greeting> getGreeting(@PathVariable("id") Long id) {
        Greeting greeting = greetingService.findOne(id);
        if(greeting == null) {
            return new ResponseEntity<Greeting>(HttpStatus.NOT_FOUND);
        }
        return new ResponseEntity<Greeting>(greeting, HttpStatus.OK);
    }

    @RequestMapping(value = "/greetings", method = RequestMethod.POST, consumes =
    MediaType.APPLICATION_JSON_VALUE, produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<Greeting> createGreeting(@RequestBody Greeting greeting) {
        Greeting savedGreeting = greetingService.create(greeting);
        return new ResponseEntity<Greeting>(savedGreeting, HttpStatus.CREATED);
    }

    @RequestMapping(value = "/greetings/{id}", method = RequestMethod.PUT, consumes =
    MediaType.APPLICATION_JSON_VALUE, produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<Greeting> updateGreeting(@PathVariable("id") Long id, @RequestBody
    Greeting greeting) {
        Greeting updatedGreeting = null;
        if (greeting != null && id == greeting.getId()) {
            updatedGreeting = greetingService.update(greeting);
        }
    }
}
```

```

    }
    if(updatedGreeting == null) {
        return new ResponseEntity<Greeting>(HttpStatus.INTERNAL_SERVER_ERROR);
    }
    return new ResponseEntity<Greeting>(updatedGreeting, HttpStatus.OK);
}

@RequestMapping(value = "/greetings/{id}", method = RequestMethod.DELETE)
public ResponseEntity<Greeting> deleteGreeting(@PathVariable("id") Long id) {
    greetingService.delete(id);
    return new ResponseEntity<Greeting>(HttpStatus.NO_CONTENT);
}
}

```

Archivo de propiedades de la aplicación para la base de datos MySQL

```

#mysql config
spring.datasource.url=jdbc:mysql://localhost:3306/springboot
spring.datasource.username=root
spring.datasource.password=Welcome@123
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
spring.jpa.hibernate.ddl-auto = update

spring.jpa.hibernate.naming-strategy=org.hibernate.cfg.DefaultNamingStrategy

#initialization
spring.datasource.schema=classpath:/data/schema.sql

```

Archivo SQL

```

drop table if exists greeting;
create table greeting (
    id bigint not null auto_increment,
    text varchar(100) not null,
    primary key(id)
);

```

archivo pom.xml

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>org</groupId>
<artifactId>springboot</artifactId>
<version>0.0.1-SNAPSHOT</version>

```

```

<packaging>war</packaging>
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.2.1.RELEASE</version>
</parent>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>

  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>

```

Construyendo un JAR ejecutable

Puede ejecutar la aplicación desde la línea de comandos con Maven. O puede crear un solo archivo JAR ejecutable que contenga todas las dependencias, clases y recursos necesarios y ejecutarlo. Esto facilita el envío, la versión y la implementación del servicio como una aplicación a lo largo del ciclo de vida del desarrollo, en diferentes entornos, etc.

Ejecute la aplicación utilizando `./mvnw spring-boot:run`. O puede construir el archivo JAR con `./mvnw clean package`. Luego puedes ejecutar el archivo JAR:

```
java -jar target/springboot-0.0.1-SNAPSHOT.jar
```

Lea Spring boot + Spring Data JPA en línea: <https://riptutorial.com/es/spring-boot/topic/6203/spring-boot-plus-spring-data-jpa>

Capítulo 18: Spring Boot- Hibernate-REST Integration

Examples

Añadir soporte Hibernate

1. Agregue la dependencia **spring-boot-starter-data-jpa** a pom.xml. Puede omitir la etiqueta de la **versión** , si está utilizando **spring-boot-starter-parent** como padre de su **pom.xml** . La dependencia a continuación trae a Hibernate y todo lo relacionado con JPA a su proyecto ([referencia](#)).

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

2. Añadir controlador de base de datos a **pom.xml** . Este a continuación es para la base de datos H2 ([referencia](#)).

```
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
</dependency>
```

3. Habilitar el registro de depuración para Hibernate en **application.properties**

logging.level.org.hibernate.SQL = debug

o en **application.yml**

```
logging:
  level:
    org.hibernate.SQL: debug
```

4. Agregue la clase de entidad al paquete deseado en **\$ {project.home} / src / main / java /** , por ejemplo, en **com.example.myproject.domain** ([referencia](#)):

```
package com.example.myproject.domain;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import java.io.Serializable;

@Entity
public class City implements Serializable {
```

```

    @Id
    @GeneratedValue
    public Long id;

    @Column(nullable = false)
    public String name;
}

```

5. Agregue **import.sql** a **\$ {project.home} / src / main / resources /** . Ponga las **instrucciones INSERT** en el archivo. Este archivo se utilizará para la población de esquemas de base de datos en cada inicio de la aplicación ([referencia](#)):

```

insert into city(name) values ('Brisbane');

insert into city(name) values ('Melbourne');

```

6. Agregue la clase de Repositorio al paquete deseado en **\$ {project.home} / src / main / java /** , por ejemplo, en **com.example.myproject.service** ([referencia](#)):

```

package com.example.myproject.service;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import java.io.Serializable;

import com.example.myproject.domain.City;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.repository.Repository;

interface CityRepository extends Repository<City, Long> {

    Page<City> findAll(Pageable pageable);

    Page<City> findByName(String name);
}

```

Básicamente eso es todo! En este punto, ya puede acceder a la base de datos utilizando los métodos de **com.example.myproject.service.CityRepository** .

Añadir soporte REST

1. Agregue la dependencia de **spring-boot-starter-web** a pom.xml. Puede omitir la etiqueta de la **versión** , si está usando **spring-boot-starter-parent** como padre de su **pom.xml** ([referencia](#)).

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>

```


2. Agregue el controlador REST al paquete deseado, por ejemplo a **com.example.myproject.web.rest** ([referencia](#)):

```
package com.example.myproject.web.rest;

import java.util.Map;
import java.util.HashMap;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import javax.servlet.http.HttpServletRequest;

@RestController
public class VersionController {
    @RequestMapping("/api/version")
    public ResponseEntity get() {
        final Map<String, String> responseParams = new HashMap();
        responseParams.put("requestStatus", "OK");
        responseParams.put("version", "0.1-SNAPSHOT");

        return ResponseEntity.ok().body(responseParams.build());
    }
}
```

3. Inicie la aplicación Spring Boot ([referencia](#)).
4. Puede acceder a su controlador en la dirección [http: // localhost: 8080 / api / version](http://localhost:8080/api/version) .

Lea Spring Boot- Hibernate-REST Integration en línea: <https://riptutorial.com/es/spring-boot/topic/6818/spring-boot--hibernate-rest-integration>

Capítulo 19: Spring-Boot + JDBC

Introducción

Spring Boot se puede utilizar para crear y conservar una base de datos relacional de SQL. Puede elegir conectarse a un H2 en la base de datos de memoria usando Spring Boot, o tal vez optar por conectarse a la base de datos MySQL, es completamente su elección. Si desea realizar operaciones CRUD contra su base de datos, puede hacerlo utilizando el bean JdbcTemplate, este bean se proporcionará automáticamente mediante Spring Boot. Spring Boot le ayudará proporcionando la configuración automática de algunos beans de uso común relacionados con JDBC.

Observaciones

Para comenzar, en su eclipse de pts vaya a nuevo -> Spring Starter Project -> complete sus coordenadas de Maven -> y agregue las siguientes dependencias:

En la pestaña SQL -> agregar JDBC + agregar MySQL (si MySQL es su elección).

Para MySQL, también deberá agregar el MySQL Java Connector.

En su archivo de aplicaciones Spring Boot application.properties (su archivo de configuración Spring Boot) necesitará configurar sus credenciales de fuente de datos para MySQL DB:

1. spring.datasource.url
2. spring.datasource.username
3. spring.datasource.password
4. spring.datasource.driver-class-name

por ejemplo:

```
spring.datasource.url=jdbc:mysql://localhost/test
spring.datasource.username=dbuser
spring.datasource.password=dbpass
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

Debajo de la carpeta de recursos agregue los siguientes dos archivos:

1. schema.sql -> cada vez que ejecute su aplicación Spring Boot ejecutará este archivo, dentro de este se supone que debe escribir su esquema de base de datos, definir tablas y sus relaciones.
2. data.sql -> cada vez que ejecute su aplicación Spring Boot ejecutará este archivo, dentro de él, se supone que debe escribir datos que se insertarán en su tabla como una inicialización inicial.

Spring Boot te proporcionará el bean JdbcTemplate automáticamente para que puedas usarlo

instantáneamente de esta manera:

```
@Autowired
private JdbcTemplate template;
```

Sin ninguna otra configuración.

Examples

archivo schema.sql

```
CREATE SCHEMA IF NOT EXISTS `backgammon`;
USE `backgammon`;

DROP TABLE IF EXISTS `user_in_game_room`;
DROP TABLE IF EXISTS `game_users`;
DROP TABLE IF EXISTS `user_in_game_room`;

CREATE TABLE `game_users`
(
  `user_id` BIGINT NOT NULL AUTO_INCREMENT,
  `first_name` VARCHAR(255) NOT NULL,
  `last_name` VARCHAR(255) NOT NULL,
  `email` VARCHAR(255) NOT NULL UNIQUE,
  `user_name` VARCHAR(255) NOT NULL UNIQUE,
  `password` VARCHAR(255) NOT NULL,
  `role` VARCHAR(255) NOT NULL,
  `last_updated_date` DATETIME NOT NULL,
  `last_updated_by` BIGINT NOT NULL,
  `created_date` DATETIME NOT NULL,
  `created_by` BIGINT NOT NULL,
  PRIMARY KEY(`user_id`)
);

DROP TABLE IF EXISTS `game_rooms`;

CREATE TABLE `game_rooms`
(
  `game_room_id` BIGINT NOT NULL AUTO_INCREMENT,
  `name` VARCHAR(255) NOT NULL,
  `private` BIT(1) NOT NULL,
  `white` BIGINT DEFAULT NULL,
  `black` BIGINT DEFAULT NULL,
  `opened_by` BIGINT NOT NULL,
  `speed` BIT(3) NOT NULL,
  `last_updated_date` DATETIME NOT NULL,
  `last_updated_by` BIGINT NOT NULL,
  `created_date` DATETIME NOT NULL,
  `created_by` BIGINT NOT NULL,
  `token` VARCHAR(255) AS (SHA1(CONCAT(`name`, "This is a qwe secret 123", `created_by`,
`created_date`))),
  PRIMARY KEY(`game_room_id`)
);

CREATE TABLE `user_in_game_room`
(
  `user_id` BIGINT NOT NULL,
```

```

`game_room_id` BIGINT NOT NULL,
`last_updated_date` DATETIME NOT NULL,
`last_updated_by` BIGINT NOT NULL,
`created_date` DATETIME NOT NULL,
`created_by` BIGINT NOT NULL,
PRIMARY KEY(`user_id`, `game_room_id`),
FOREIGN KEY (`user_id`) REFERENCES `game_users` (`user_id`),
FOREIGN KEY (`game_room_id`) REFERENCES `game_rooms` (`game_room_id`)
);

```

Primera aplicación de arranque JdbcTemplate

```

@SpringBootApplication
@RestController
public class SpringBootJdbcApplication {

    @Autowired
    private JdbcTemplate template;

    @RequestMapping("/cars")
    public List<Map<String, Object>> stocks() {
        return template.queryForList("select * from car");
    }

    public static void main(String[] args) {
        SpringApplication.run(SpringBootJdbcApplication.class, args);
    }
}

```

data.sql

```

insert into game_users values(..., ..., ..., ...);
insert into game_users values(..., ..., ..., ...);
insert into game_users values(..., ..., ..., ...);

```

Lea Spring-Boot + JDBC en línea: <https://riptutorial.com/es/spring-boot/topic/9834/spring-boot-plus-jdbc>

Capítulo 20: ThreadPoolTaskExecutor: configuración y uso

Examples

configuración de la aplicación

```
@Configuration
@EnableAsync
public class ApplicationConfiguration{

    @Bean
    public TaskExecutor getAsyncExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(2);
        executor.setThreadNamePrefix("executor-task-");
        executor.initialize();
        return executor;
    }
}
```

Lea ThreadPoolTaskExecutor: configuración y uso en línea: <https://riptutorial.com/es/spring-boot/topic/7497/threadpooltaskexecutor--configuracion-y-uso>

Creditos

S. No	Capítulos	Contributors
1	Comenzando con el arranque de primavera	Andy Wilkinson , Brice Roncace , Community , GVArt , imdzeeshan , ipsi , kodiak , loki2302 , M. Deinum , Marvin Frommhold , Matthew Fontana , MeysaM , Misa Lazovic , Moshiour , Nikem , pinkpanther , rajadilipkolli , RamenChef , Ronnie Wang , Slava Semushin , Szobi , Tom
2	Almacenamiento en caché con Redis usando Spring Boot para MongoDB	rajadilipkolli
3	Aplicación web de arranque Spring-Responsive con JHipster	anataliocs , codependent
4	Bota de primavera + JPA + mongoDB	Andy Wilkinson , Matthew Fontana , Rakesh , RubberDuck , Stephen Leppik
5	Bota de Primavera + JPA + RESTO	incomplete-co.de
6	Conectando una aplicación de arranque de primavera a MySQL	koder23 , sunkuet02
7	Controladores	Amit Gujarathi
8	Creación y uso de múltiples archivos de aplicaciones.propiedades	Patrick
9	Escaneo de paquetes	Tom
10	Implementación de la aplicación de ejemplo utilizando Spring-boot en Amazon Elastic Beanstalk	Praveen Kumar K S
11	Instalación de la CLI de inicio de Spring	Robert Thornton
12	Microservicio de arranque	Matthew Fontana

de primavera con JPA		
13	Pruebas en Spring Boot	Ali Dehghani , Aman Tuladhar , rajadilipkolli , Slava Semushin
14	Servicios de descanso	Andy Wilkinson , CAPS LOCK , g00glen00b , Johir , Kevin Wittek , Manish Kothari , odedia , Ronnie Wang , Safwan Hijazi , shyam , Soner
15	Spring boot + Hibernate + Web UI (Thymeleaf)	ppeterka , rajadilipkolli , Tom
16	Spring Boot + Spring Data Elasticsearch	sunkuet02
17	Spring boot + Spring Data JPA	dunni , Johir , naXa , Sanket , Sohlowmawn , sunkuet02
18	Spring Boot- Hibernate- REST Integration	Igor Bljahhin
19	Spring-Boot + JDBC	Moshe Arad
20	ThreadPoolTaskExecutor: configuración y uso	Rosteach