eBook Gratuit

APPRENEZ spring-boot

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#spring-boot

Table des matières

À propos

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.

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

Chapitre 1: Commencer avec Spring-Boot

Remarques

Cette section fournit une vue d'ensemble de ce que Spring-Boot est et pourquoi un développeur peut vouloir l'utiliser.

Il devrait également mentionner tout sujet important dans le cadre du programme Spring-Boot et établir un lien avec les sujets connexes. Comme la documentation de Spring-Boot est nouvelle, vous devrez peut-être créer des versions initiales de ces rubriques connexes.

Versions

Version	Date de sortie
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

Installation ou configuration

L'installation avec Spring Boot pour la première fois est assez rapide grâce au travail acharné de la communauté Spring.

Conditions préalables:

1. Java installé
2. Java IDE recommandé non requis (Intellij, Eclipse, Netbeans, etc.)

Vous n'avez pas besoin d'installer Maven et / ou Gradle. Les projets générés par Spring Initializr sont fournis avec un Maven Wrapper (commande `mvnw`) ou un Gradle Wrapper (commande `gradlew`).

Ouvrez votre navigateur Web sur https://start.spring.io Ceci est un tableau de bord pour la création de nouvelles applications Spring Boot pour le moment, nous allons aller avec le strict

minimum.

N'hésitez pas à passer de Maven à Gradle si c'est votre outil de construction préféré.

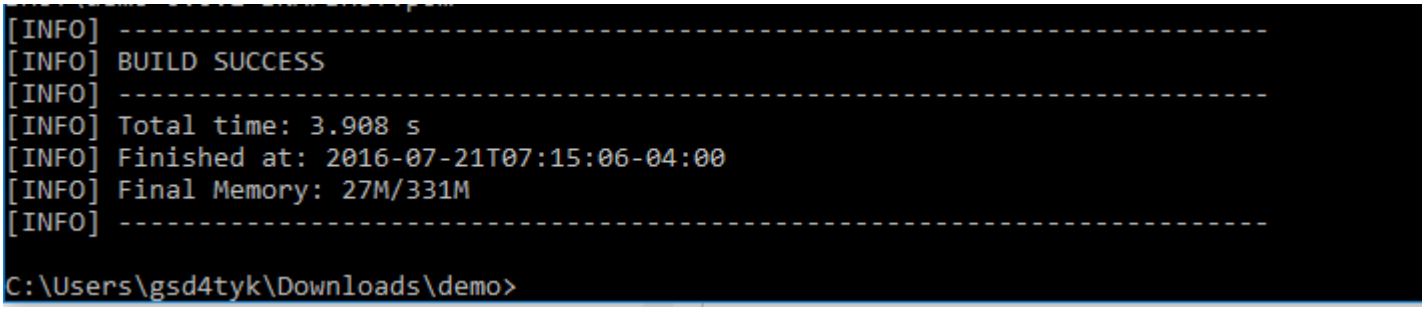
Recherchez "Web" sous "Rechercher les dépendances" et ajoutez-le.

Cliquez sur Générer un projet!

Cela va télécharger un fichier zip appelé démo N'hésitez pas à extraire ce fichier où vous voulez sur votre ordinateur.

Si vous sélectionnez maven, naviguez dans une invite de commande vers le répertoire de base et `mvn clean install` commande `mvn clean install`

Vous devriez obtenir un résultat de construction réussi:



Exécuter votre application: `mvn spring-boot:run`

Votre application Spring Boot démarre maintenant. Naviguez dans votre navigateur Web vers localhost: 8080

Félicitations! Vous venez de lancer votre première application Spring Boot. Maintenant, ajoutons un tout petit peu de code pour que vous puissiez le voir fonctionner.

Donc, utilisez `ctrl` + `c` pour quitter votre serveur en cours d'exécution.

Accédez à: `src/main/java/com/example/DemoApplication.java` Mettez à jour cette classe pour avoir un contrôleur

```
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);
    }
}
```

De bonnes choses permettent maintenant de construire et d'exécuter à nouveau le projet avec `mvn clean install spring-boot:run` !

Naviguez maintenant dans votre navigateur Web pour localiser l'hôte: 8080

Bonjour le monde!

Félicitations! Nous venons de terminer la création d'une application d'amorçage de printemps et de configurer notre premier contrôleur pour qu'il renvoie "Hello World!" Bienvenue dans le monde de Spring Boot!

Application web simple Spring Boot utilisant Gradle comme système de construction

Cet exemple suppose que vous avez déjà installé Java et Gradle .

Utilisez la structure de projet suivante:

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

`build.gradle` est votre script de génération pour le système de génération Gradle avec le contenu suivant:

```
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` est la classe principale de l'application 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!";
  }
}
```

Vous pouvez maintenant exécuter l'application Web Spring Boot avec

```
gradle bootRun
```

et accéder au point de terminaison HTTP publié en utilisant `curl`

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

ou votre navigateur en ouvrant localhost: 8080 / hello .

Lire Commencer avec Spring-Boot en ligne: https://riptutorial.com/fr/spring-boot/topic/829/commencer-avec-spring-boot

Chapitre 2: Analyse de paquet

Introduction

Dans cette rubrique, je vais donner un aperçu de la numérisation du package de démarrage au printemps.

Vous pouvez trouver des informations de base dans les documents de démarrage du printemps dans le lien suivant (en utilisant-boot-structuring-your-code) mais je vais essayer de fournir des informations plus détaillées.

Spring Boot, et Spring en général, fournissent une fonctionnalité pour analyser automatiquement les packages de certaines annotations afin de créer des `beans` et une `configuration` .

Paramètres

Annotation	Détails
@PrintBootApplication	Annotation principale de l'application de démarrage à ressort. utilisé une fois dans l'application, contient une méthode principale et sert de package principal pour l'analyse des packages
@PrintBootConfiguration	Indique qu'une classe fournit une application Spring Boot. Doit être déclaré une seule fois dans l'application, généralement automatiquement en définissant `@SpringBootApplication`
@EnableAutoConfiguration	Activer la configuration automatique du contexte d'application Spring. Doit être déclaré une seule fois dans l'application, généralement automatiquement en définissant `@SpringBootApplication`
@ CompoScan	Utilisé pour déclencher une analyse automatique des packages sur un package donné et ses enfants ou pour définir une analyse de package personnalisée
@Configuration	Utilisé pour déclarer une ou plusieurs méthodes `@Bean` . Peut être sélectionné par analyse automatique des packages afin de déclarer une ou plusieurs méthodes `@Bean` au lieu d'une configuration XML classique
@Haricot	Indique qu'une méthode produit un bean à gérer par le conteneur Spring. En `@Bean` générale, les méthodes annotées `@Bean` seront placées dans les classes annotées `@Configuration` qui seront sélectionnées par l'analyse du package pour créer des beans basés sur la configuration Java.

Annotation	Détails
@Composant	En déclarant une classe en tant que `@Component` elle devient un candidat à la détection automatique lors de l'utilisation de la configuration basée sur des annotations et de l'analyse des classpath. En général, une classe annotée avec `@Component` deviendra un `bean` dans l'application
@Dépôt	Défini à l'origine par Domain-Driven Design (Evans, 2003) comme "un mécanisme d'encapsulation du stockage. Il est généralement utilisé pour indiquer un `Repository` de `spring data`
@Un service	Très similaire en pratique à `@Component` . Défini à l'origine par Domain-Driven Design (Evans, 2003) comme "une opération offerte en tant qu'interface autonome dans le modèle, sans état encapsulé".
@Manette	Indique qu'une classe annotée est un "Controller" (par exemple un contrôleur Web).
@RestController	Une annotation pratique qui est elle-même annotée avec `@Controller` et `@ResponseBody` . Sera automatiquement sélectionné par défaut car il contient l'annotation `@Controller` qui est sélectionnée par défaut.

Examples

@PrintBootApplication

Le moyen le plus simple de structurer votre code à l'aide du démarrage par ressort pour une analyse automatique de package `@SpringBootApplication` consiste à utiliser l'annotation `@SpringBootApplication` . Cette annotation fournit en elle-même 3 autres annotations `@SpringBootConfiguration` analyse automatique: `@SpringBootConfiguration` , `@EnableAutoConfiguration` , `@ComponentScan` (plus d'informations sur chaque annotation dans la section `Parameters`).

`@SpringBootApplication` sera `@SpringBootApplication` placé dans le package principal et tous les autres composants seront placés dans des packages sous ce fichier:

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

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

Sauf indication contraire, le démarrage du printemps détecte automatiquement les annotations `@Configuration` , `@Component` , `@Repository` , `@Service` , `@Controller` et `@RestController` sous les packages analysés (`@Configuration` et `@RestController` sont sélectionnés car annotés par `@Component` et `@Controller` conséquence)).

Exemple de code de base:

```
@SpringBootApplication
public class Application {

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

}
```

Définition explicite de paquets / classes

Depuis la version **1.3,** vous pouvez également indiquer que le démarrage du printemps analyse des packages spécifiques en définissant `scanBasePackages` ou `scanBasePackageClasses` dans `@SpringBootApplication` au lieu de spécifier `@ComponentScan` .

1. `@SpringBootApplication(scanBasePackages = "com.example.myproject")` : définit `com.example.myproject` comme package de base à analyser.
2. `@SpringBootApplication(scanBasePackageClasses = CustomerController.class)` - une alternative sécurisée à `scanBasePackages` définit le package de `CustomerController.java` , `com.example.myproject.web` , comme package de base à analyser.

Hors configuration automatique

Une autre fonctionnalité importante est la possibilité d'exclure des classes de configuration automatique spécifiques à l'aide de `exclude` ou `excludeName` (`excludeName` existe depuis la version **1.3**).

1. `@SpringBootApplication(exclude = DemoConfiguration.class)` - exclura `DemoConfiguration` de l'analyse automatique des packages.
2. `@SpringBootApplication(excludeName = "DemoConfiguration")` - fera de même en utilisant le nom de classe entièrement classé.

@ CompoScan

Vous pouvez utiliser `@ComponentScan` pour configurer une analyse de package plus complexe. Il y a aussi `@ComponentScans` qui agit comme une annotation de conteneur qui regroupe plusieurs annotations `@ComponentScan` .

Exemples de code de base

```
@ComponentScan
public class DemoAutoConfiguration {
}
```

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

`@ComponentScan` sans configuration agit comme `@SpringBootApplication` et analyse tous les packages sous la classe annotée avec cette annotation.

Dans cet exemple, je vais indiquer certains des attributs utiles de `@ComponentScan` :

1. **basePackages** - peut être utilisé pour indiquer des packages spécifiques à analyser.
2. **useDefaultFilters** - En définissant cet attribut sur false (par défaut true), vous pouvez vous assurer que Spring `@Component` pas `@Component` , `@Repository` , `@Service` ou `@Controller` .
3. **includeFilters** - peut être utilisé pour *inclure* des annotations / motifs de regex de printemps spécifiques à inclure dans l'analyse des packages.
4. **excludeFilters** - peut être utilisé pour *exclure* des motifs d'annotation / regex de printemps spécifiques à inclure dans l'analyse de package.

Il existe beaucoup plus d'attributs, mais ceux-ci sont les plus couramment utilisés pour personnaliser l'analyse des packages.

Créer votre propre configuration automatique

Spring boot est basé sur de nombreux projets parent pré-configurés. Vous devriez déjà être familier avec les projets de démarrage de printemps.

Vous pouvez facilement créer votre propre projet de démarrage en procédant comme suit:

1. Créez des classes `@Configuration` pour définir les beans par défaut. Vous devez utiliser autant que possible les propriétés externes pour autoriser la personnalisation et essayer d'utiliser des annotations d'assistance automatique telles que `@AutoConfigureBefore` , `@AutoConfigureAfter` , `@ConditionalOnBean` , `@ConditionalOnMissingBean` etc. Vous trouverez des informations plus détaillées sur chaque annotation dans la documentation officielle .
2. Placez un fichier / fichier de configuration automatique qui regroupe toutes les classes `@Configuration` .
3. Créez un fichier nommé `spring.factories` et placez-le dans `src/main/resources/META-INF` .
4. Dans `spring.factories` , définissez la propriété `org.springframework.boot.autoconfigure.EnableAutoConfiguration` avec les valeurs séparées par des virgules de vos classes `@Configuration` :

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

En utilisant cette méthode, vous pouvez créer vos propres classes de configuration automatique

qui seront sélectionnées par boot-spring. Spring-boot analyse automatiquement toutes les dépendances maven / gradle pour un fichier `spring.factories` , s'il en trouve un, il ajoute toutes les classes `@Configuration` spécifiées dans son processus de configuration automatique.

Assurez-vous que votre projet d' `spring boot maven plugin auto-configuration` ne contient pas de `spring boot maven plugin` - `spring boot maven plugin` car il compilera le projet en tant que fichier JAR exécutable et ne sera pas chargé par le chemin de `spring.factories` comme `spring.factories` . ne chargera pas votre configuration

Lire Analyse de paquet en ligne: https://riptutorial.com/fr/spring-boot/topic/9354/analyse-de-paquet

Chapitre 3: Application Web Spring Boot entièrement réactive avec JHipster

Examples

Créer une application Spring Boot en utilisant jHipster sur Mac OS

jHipster vous permet de démarrer une application Web Spring Boot avec un back-end API REST et un front-end AngularJS et Twitter Bootstrap.

Plus sur jHipster ici: jHipster Documentation

Installez l'infusion:

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

Voir des informations supplémentaires sur la façon d'installer brew ici: Installer Brew

Installer Gradle

Gradle est un système de gestion et de construction de dépendances.

```
brew install gradle
```

Installer Git

Git est un outil de contrôle de version

```
brew install git
```

Installer NodeJS

NodeJS vous donne accès à npm, le gestionnaire de paquets nodal nécessaire pour installer d'autres outils.

```
brew install node
```

Installez Yeoman

Yeoman est un générateur

```
npm install -g yo
```

Installer Bower

Bower est un outil de gestion des dépendances

```
npm install -g bower
```

Installez Gulp

Gulp est un coureur de tâche

```
npm install -g gulp
```

Installez jHipster Yeoman Generator

Ceci est le générateur jHipster

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

Créer une application

Ouvrez une fenêtre de terminal.

Accédez au répertoire racine où vous conserverez vos projets. Créez un répertoire vide dans lequel vous allez créer votre application

```
mkdir myapplication
```

Allez dans ce répertoire

```
cd myapplication/
```

Pour générer votre application, tapez

```
yo jhipster
```

Vous serez invité avec les questions suivantes

Quel type d'application souhaitez-vous créer?

Votre type d'application dépend de si vous souhaitez utiliser une architecture de microservices ou non. Une explication complète sur les microservices est disponible ici, si vous n'êtes pas sûr, utilisez l'application par défaut «Monolithic Application».

Choisissez l' *application* par défaut si vous n'êtes pas sûr

Quel est votre nom de package Java par défaut?

Votre application Java l'utilisera comme package racine.

Quel type d'authentification souhaitez-vous utiliser?

Si vous n'êtes pas sûr d'utiliser la *sécurité Spring de* base basée sur la session par défaut

Quel type de base de données souhaitez-vous utiliser?

Quelle base de développement souhaitez-vous utiliser?

C'est la base de données que vous utiliserez avec votre profil «développement». Vous pouvez soit utiliser:

Utilisez H2 par défaut si vous n'êtes pas sûr

H2, en cours d'exécution en mémoire. C'est le moyen le plus simple d'utiliser JHipster, mais vos données seront perdues au redémarrage de votre serveur.

Voulez-vous utiliser le cache de deuxième niveau Hibernate?

Hibernate est le fournisseur JPA utilisé par JHipster. Pour des raisons de performances, nous vous recommandons fortement d'utiliser un cache et de l'adapter aux besoins de votre application. Si vous choisissez de le faire, vous pouvez utiliser soit ehcache (cache local), soit Hazelcast (cache distribué, à utiliser dans un environnement en cluster).

Voulez-vous utiliser un moteur de recherche dans votre application? Elasticsearch sera configuré à l'aide de Spring Data Elasticsearch. Vous pouvez trouver plus d'informations sur notre guide Elasticsearch.

Choisissez non si vous n'êtes pas sûr

Voulez-vous utiliser des sessions HTTP en cluster?

Par défaut, JHipster utilise une session HTTP uniquement pour stocker les informations d'authentification et d'autorisation de Spring Security. Bien entendu, vous pouvez choisir de mettre plus de données dans vos sessions HTTP. L'utilisation de sessions HTTP entraînera des problèmes si vous exécutez dans un cluster, en particulier si vous n'utilisez pas un équilibreur de charge avec des «sessions sticky». Si vous souhaitez répliquer vos sessions dans votre cluster, choisissez cette option pour configurer Hazelcast.

Choisissez non si vous n'êtes pas sûr

Voulez-vous utiliser WebSockets? Les Websockets peuvent être activés à l'aide de Spring Websocket. Nous fournissons également un échantillon complet pour vous montrer comment utiliser efficacement le cadre.

Choisissez non si vous n'êtes pas sûr

Voulez-vous utiliser Maven ou Gradle? Vous pouvez créer votre application Java générée avec Maven ou Gradle. Maven est plus stable et plus mature. Gradle est plus flexible, plus facile à étendre et plus dynamique.

Choisissez *Gradle* si vous n'êtes pas sûr

Voulez-vous utiliser le préprocesseur de feuille de style LibSass pour votre CSS? Node-sass est une excellente solution pour simplifier la conception de CSS. Pour être utilisé efficacement, vous devrez exécuter un serveur Gulp, qui sera configuré automatiquement.

Choisissez non si vous n'êtes pas sûr

Souhaitez-vous activer le support de traduction avec Angular Translate? Par défaut, JHipster fournit une excellente prise en charge de l'internationalisation, à la fois du côté client avec Angular Translate et du côté serveur. Cependant, l'internationalisation ajoute un peu de temps et est un peu plus complexe à gérer, vous pouvez donc choisir de ne pas installer cette fonctionnalité.

Choisissez non si vous n'êtes pas sûr

Quels frameworks de test souhaitez-vous utiliser? Par défaut, JHipster fournit des tests d'unité / d'intégration Java (utilisant le support JUnit de Spring) et des tests unitaires JavaScript (utilisant Karma.js). En option, vous pouvez également ajouter un support pour:

Ne choisissez rien si vous n'êtes pas sûr. Vous aurez accès au junit et au karma par défaut.

Lire Application Web Spring Boot entièrement réactive avec JHipster en ligne: https://riptutorial.com/fr/spring-boot/topic/6297/application-web-spring-boot-entierement-reactive-avec-jhipster

Chapitre 4: Botte de printemps + JPA + mongoDB

Examples

Opération CRUD à MongoDB avec JPA

Modèle client

```
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;
    }

}
```

Contrôleur client

```
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.finedById(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);
    }
}
```

Référentiel client

```
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

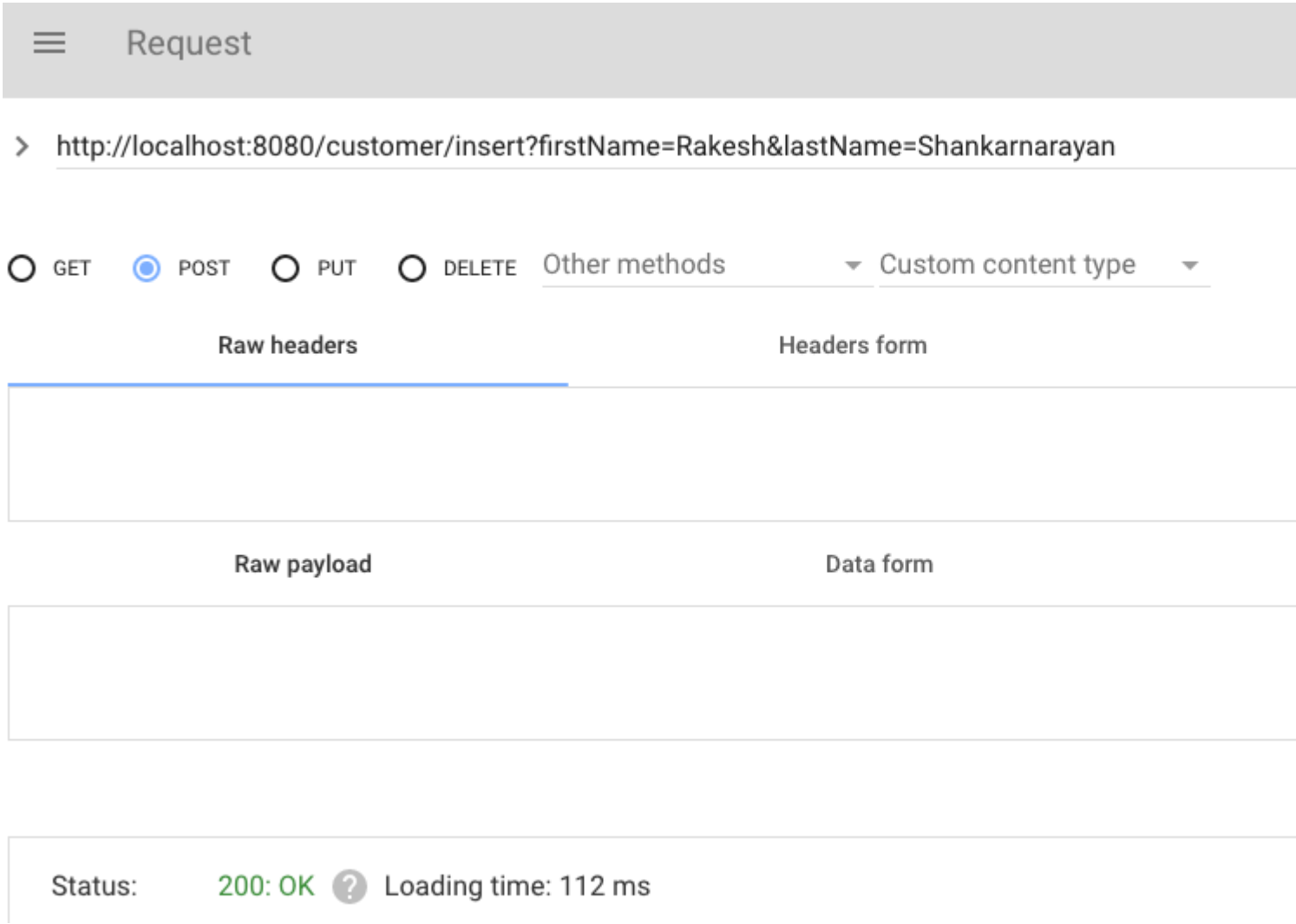
Veuillez ajouter ci-dessous les dépendances dans le fichier pom.xml:

```
<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>
```

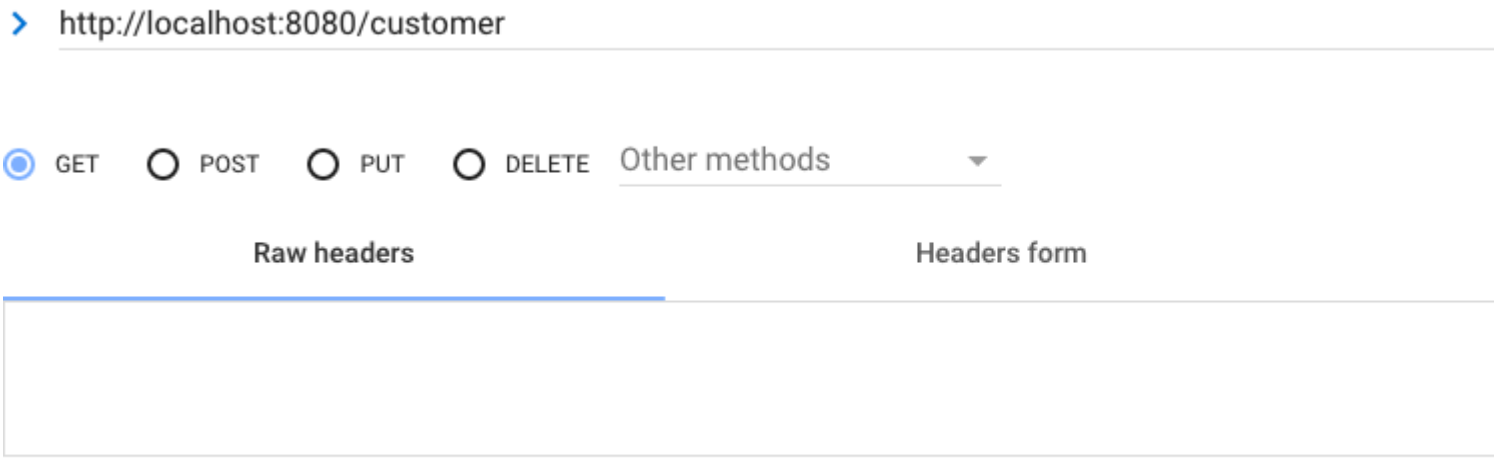
Insérer des données à l'aide du client de repos: méthode POST

Pour tester notre application, j'utilise un client de repos avancé qui est une extension chrome:

Donc, voici l'instantané pour insérer les données:



Obtenir une URL de demande



Obtenir le résultat de la demande:

Lire Botte de printemps + JPA + mongoDB en ligne: https://riptutorial.com/fr/spring-boot/topic/3398/botte-de-printemps-plus-jpa-plus-mongodb

Chapitre 5: Botte de printemps + JPA + REST

Remarques

Cet exemple utilise Spring Boot, Spring Data JPA et Spring Data REST pour exposer un objet de domaine géré par JPA simple via REST. L'exemple répond avec le format JAL HAL et expose une URL accessible sur `/person` . La configuration Maven comprend une base de données H2 en mémoire pour prendre en charge une mise en veille rapide.

Examples

Spring Boot Startup

```
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);
    }

}
```

Objet de domaine

```
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;
    }

}
```

Interface de référentiel

```
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
RestResouce(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) { }


}
```

Configuration 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>
```

Lire Botte de printemps + JPA + REST en ligne: https://riptutorial.com/fr/spring-boot/topic/6507/botte-de-printemps-plus-jpa-plus-rest

Chapitre 6: Botte de printemps + Spring Data JPA

Introduction

Spring Boot facilite la création d'applications et de services de qualité à l'aide de Spring avec un minimum d'effets. Il privilégie la convention sur la configuration.

Spring Data JPA , qui fait partie de la famille Spring Data , facilite l'implémentation de référentiels basés sur JPA. Cela facilite la création d'applications utilisant des technologies d'accès aux données.

Remarques

Annotations

`@Repository` : Indique qu'une classe annotée est un "référentiel", un mécanisme permettant d'encapsuler le comportement de stockage, de récupération et de recherche qui émule une collection d'objets. Les équipes implémentant des modèles J2EE traditionnels, tels que «Data Access Object», peuvent également appliquer ce stéréotype aux classes DAO, mais il convient de prendre soin de bien comprendre la distinction entre les référentiels Data Access Object et DDD. Cette annotation est un stéréotype d'usage général et les équipes individuelles peuvent restreindre leur sémantique et les utiliser comme il convient.

`@RestController` : Une annotation pratique qui est lui - même annotés avec `@Controller` et `@ResponseBody.Types` qui portent cette annotation sont traités comme des contrôleurs où `@RequestMapping` méthodes supposent `@ResponseBody` sémantique par défaut.

`@Service` : Indique qu'une classe annotée est un "Service" (par exemple une façade de service métier). Cette annotation sert de spécialisation de `@Component` , permettant aux classes d'implémentation d'être automatiquement détectées via l'analyse des `@Component` de classes.

`@SpringBootApplication` : de nombreux développeurs Spring Boot ont toujours leur classe principale annotée avec `@Configuration` , `@EnableAutoConfiguration` et `@ComponentScan` . Comme ces annotations sont fréquemment utilisées ensemble (surtout si vous suivez les meilleures pratiques ci-dessus), Spring Boot fournit une alternative pratique à `@SpringBootApplication` .

`@Entity` : Spécifie que la classe est une entité. Cette annotation est appliquée à la classe d'entité.

Documentation officielle

Pivotal Software a fourni une documentation assez complète sur Spring Framework, qui se trouve

à l'adresse suivante:

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

Examples

Exemple de base de l'intégration Spring Spring et Spring Data JPA

Nous allons créer une application qui stocke les POJO dans une base de données. L'application utilise Spring Data JPA pour stocker et récupérer des données dans une base de données relationnelle. Sa fonction la plus convaincante est la possibilité de créer automatiquement des implémentations de référentiel, à l'exécution, à partir d'une interface de référentiel.

Classe principale

```
package org.springboot;

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);
    }
}
```

La méthode `main()` utilise la méthode `SpringApplication.run()` Spring Boot pour lancer une application. Veuillez noter qu'il n'y a pas une seule ligne de XML. Pas de fichier web.xml non plus. Cette application Web est 100% Java pur et vous n'avez pas à gérer la configuration de la plomberie ou de l'infrastructure.

Classe d'entité

```
package org.springboot.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. */
}
```

Ici vous avez une classe `Greeting` avec deux attributs, l' `id` et le `text` . Vous avez également deux constructeurs. Le constructeur par défaut n'existe que pour JPA. Vous ne l'utiliserez pas directement, il peut donc être désigné comme `protected` . L'autre constructeur est celui que vous utiliserez pour créer des instances de `Greeting` d' `Greeting` à enregistrer dans la base de données.

La classe de `@Entity` `Greeting` est annotée avec `@Entity` , indiquant qu'il s'agit d'une entité JPA. En l'absence d'une annotation `@Table` , il est supposé que cette entité sera mappée sur une table nommée 'Greeting'.

La propriété `id` `@Id` d' `@Id` est annotée avec `@Id` afin que JPA le reconnaisse comme identifiant de l'objet. La propriété `id` est également annotée avec `@GeneratedValue` pour indiquer que l'ID doit être généré automatiquement.

L'autre propriété, `text` est laissée non annotée. Il est supposé qu'il sera mappé à une colonne qui partage le même nom que la propriété elle-même.

Propriétés transitoires

Dans une classe d'entité similaire à celle ci-dessus, nous pouvons avoir des propriétés que nous ne voulons pas conserver dans la base de données ou des colonnes dans notre base de données, peut-être parce que nous voulons simplement les définir à l'exécution et les utiliser dans notre application. nous pouvons donc avoir cette propriété annotée avec l'annotation @Transient.

```
package org.springboot.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. */
}
```

Vous avez ici la même classe de `textInSomeLanguage` accueil qui a maintenant une propriété transitoire `textInSomeLanguage` qui peut être initialisée et utilisée à l'exécution et ne sera pas conservée dans la base de données.

Classe DAO

```
package org.springboot.repository;

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

public interface GreetingRepository extends CrudRepository<Greeting, Long> {

    List<Greeting> findByText(String text);
}
```

`GreetingRepository` étend l'interface `CrudRepository` . Le type d'entité et l'ID avec lesquels il fonctionne, `Greeting` et `Long` , sont spécifiés dans les paramètres génériques de `CrudRepository` . En étendant `CrudRepository` , `GreetingRepository` hérite de plusieurs méthodes pour utiliser la persistance `Greeting` , y compris des méthodes pour enregistrer, supprimer et rechercher des entités de `CrudRepository` `Greeting` .
Voir cette discussion pour la comparaison de `CrudRepository` , `PagingAndSortingRepository` , `JpaRepository` .

Spring Data JPA permet également de définir d'autres méthodes de requête en déclarant simplement leur signature de méthode. Dans le cas de `GreetingRepository` , cela est montré avec une méthode `findByText()` .

Dans une application Java typique, vous vous attendez à écrire une classe qui implémente `GreetingRepository` . Mais c'est ce qui fait la puissance de Spring Data JPA: vous n'avez pas besoin d'écrire une implémentation de l'interface du référentiel. Spring Data JPA crée une implémentation à la volée lorsque vous exécutez l'application.

Classe de service

```
package org.springboot.service;

import java.util.Collection
```

```
import org.springboot.model.Greeting;

public interface GreetingService {

    Collection<Greeting> findAll();
    Greeting findOne(Long id);
    Greeting create(Greeting greeting);
    Greeting update(Greeting greeting);
    void delete(Long id);

}
```

Service Bean

```
package org.springboot.service;

import java.util.Collection;
import org.springboot.model.Greeting;
import org.springboot.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);
    }

}
```

Classe de contrôleur

```
package org.springboot.web.api;

import java.util.Collection;
import org.springboot.model.Greeting;
import org.springboot.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);
    }

}
```

Fichier de propriétés d'application pour la base de données 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
```

Fichier SQL

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

fichier 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>
```

Construire un fichier JAR exécutable

Vous pouvez exécuter l'application à partir de la ligne de commande avec Maven. Vous pouvez également créer un seul fichier JAR exécutable contenant toutes les dépendances, classes et ressources nécessaires et l'exécuter. Cela facilite l'envoi, la version et le déploiement du service en tant qu'application tout au long du cycle de développement, dans différents environnements, etc.

Exécutez l'application en utilisant `./mvnw spring-boot:run` . Ou vous pouvez créer le fichier JAR avec le `./mvnw clean package` . Ensuite, vous pouvez exécuter le fichier JAR:

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

Lire Botte de printemps + Spring Data JPA en ligne: https://riptutorial.com/fr/spring-

boot/topic/6203/botte-de-printemps-plus-spring-data-jpa

Chapitre 7: Connecter une application Spring-Boot à MySQL

Introduction

Nous savons que Spring-Boot par défaut fonctionne avec la base de données H2. Dans cet article, nous verrons comment modifier la configuration par défaut pour utiliser la base de données MySQL.

Remarques

Pré-requis, assurez-vous que MySQL est déjà en cours d'exécution sur le port 3306 et que votre base de données est créée.

Examples

Exemple de démarrage à l'aide de MySQL

Nous suivrons le guide officiel pour Spring-Boot et Spring-Data-Jpa . Nous allons construire l'application en utilisant gradle.

1. **Créer le fichier de construction graduel**

   build.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. **Créer l'entité client**

   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. **Créer des référentiels**

   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. **Créer le fichier 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. **Créez le fichier PersistenceConfig.java**

   A l'étape 5, nous définirons comment la source de données sera chargée et comment notre application se connecte à MySQL. L'extrait ci-dessus est la configuration minimale nécessaire à la connexion à MySQL. Ici nous fournissons deux haricots:

```
@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*** Cela nous permet de gérer les configurations EntityManagerFactory et nous permet de faire des personnalisations. Cela nous permet également d'injecter le PersistenceContext dans nos composants comme ci-dessous:

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

- ***DataSource*** Ici, nous retournons une instance de `DriverManagerDataSource` . Il s'agit d'une implémentation simple de l'interface JDBC DataSource standard, qui permet de configurer un ancien pilote JDBC via les propriétés du bean, et de renvoyer une nouvelle connexion pour chaque appel getConnection. Notez que je recommande d'utiliser ceci strictement à des fins de test car il existe de meilleures alternatives comme `BasicDataSource` disponible. Reportez-vous ici pour plus de détails

6. **Créer une classe d'application**

   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. **Lancer l'application**

Si vous utilisez un IDE tel que STS , vous pouvez simplement faire un clic droit sur votre projet -> Exécuter en tant que -> Gradle (STS) Build ... Dans la liste des tâches, tapez bootRun et Run.

Si vous utilisez gradle sur la ligne de commande , vous pouvez simplement exécuter l'application comme suit:

```
./gradlew bootRun
```

Vous devriez voir quelque chose comme ceci:

```
== 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']
```

Lire Connecter une application Spring-Boot à MySQL en ligne: https://riptutorial.com/fr/spring-boot/topic/8588/connecter-une-application-spring-boot-a-mysql

Chapitre 8: Contrôleurs

Introduction

Dans cette section, je vais ajouter un exemple pour le contrôleur Spring Boot Rest avec Get et Post Request.

Examples

Contrôleur de repos de démarrage à ressort.

> Dans cet exemple, je montrerai comment formuler un contrôleur de repos pour obtenir et publier des données dans la base de données en utilisant JPA avec le code le plus simple et le plus facile.

Dans cet exemple, nous ferons référence à la table de données nommée buyerRequirement.

BuyingRequirement.java

@Entity @Table (name = "BUYINGREQUIREMENTS") @NamedQueries ({@NamedQuery (name = "BuyingRequirement.findAll", query = "SELECT b DE BuyingRequirement b")}) Classe publique BuyingRequirementène 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;
}
```

}

C'est la classe d'entité qui inclut le paramètre faisant référence aux colonnes de la table byuingRequirement et leurs getters et setters.

IBuyingRequirementsRepository.java (interface 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;
    }
}
```

Il comprend la méthode

1. Post méthode qui publie des données dans la base de données.
2. Obtenez la méthode qui récupère tous les enregistrements du tableau buyRequirement.
3. C'est aussi une méthode get qui trouvera l'exigence d'achat par le nom de la personne.

Lire Contrôleurs en ligne: https://riptutorial.com/fr/spring-boot/topic/10635/controleurs

Chapitre 9: Créer et utiliser plusieurs fichiers application.properties

Examples

Environnement Dev et Prod utilisant différentes sources de données

Après avoir configuré avec succès l'application Spring-Boot, toute la configuration est gérée dans un fichier application.properties. Vous trouverez le fichier sur `src/main/resources/` .

Normalement, il est nécessaire d'avoir une base de données derrière l'application. Pour le développement de son bon d'avoir une configuration de `dev` et un `prod` environnement. En utilisant plusieurs fichiers `application.properties` , vous pouvez indiquer à Spring-Boot quel environnement l'application doit démarrer.

Un bon exemple consiste à configurer deux bases de données. Un pour `dev` et un pour `productive` .

Pour le `dev` environnement , vous pouvez utiliser une base de données en mémoire comme `H2` . Créez un nouveau fichier dans le répertoire `src/main/resources/` nommé `application-dev.properties` . Dans le fichier se trouve la configuration de la base de données en mémoire:

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

Pour l'environnement de `prod` , nous nous connecterons à une "vraie" base de données, par exemple `postgreSQL` . Créez un nouveau fichier dans le répertoire `src/main/resources/` nommé `application-prod.properties` . Dans le fichier se trouve la configuration de la base de données `postgreSQL` :

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

Dans votre fichier `application.properties` par défaut `application.properties` vous pouvez maintenant définir quel profil est activé et utilisé par Spring-Boot. Il suffit de définir un attribut à l'intérieur:

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

ou

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

Important est que la partie after `-` in `application-dev.properties` est l'identifiant du fichier.

Vous pouvez maintenant lancer l'application Spring-Boot en mode de développement ou de production en modifiant simplement l'identifiant. Une base de données en mémoire démarrera ou la connexion à une "vraie" base de données. Bien sûr, il y a beaucoup plus de cas d'utilisation pour avoir plusieurs fichiers de propriétés.

Définissez le bon profil de ressort en créant automatiquement l'application (maven)

En créant plusieurs fichiers de propriétés pour les différents environnements ou cas d'utilisation, il est parfois difficile de modifier manuellement la valeur `active.profile` pour la valeur correcte. Mais il existe un moyen de définir le `active.profile` dans le fichier `application.properties` lors de la création de l'application à l'aide de `maven-profiles` .

Disons qu'il y a trois fichiers de propriétés d'environnement dans notre application:

`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
```

Ces trois fichiers diffèrent simplement par le nom du port et du profil actif.

Dans le fichier principal `application.properties` , nous définissons notre profil Spring en utilisant une variable maven :

`application.properties` .

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

Après cela, il suffit d'ajouter les profils `pom.xml` dans notre `pom.xml` Nous allons définir des profils pour les trois environnements:

```
 <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>
            <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>
```

Vous êtes maintenant capable de construire l'application avec maven. Si vous ne définissez pas de profil Maven, sa construction par défaut (dans cet exemple, c'est dev). Pour spécifier un, vous devez utiliser un mot-clé maven. Le mot-clé pour définir un profil dans maven est `-P` directement suivi du nom du profil: `mvn clean install -Ptest` .

Maintenant, vous pouvez également créer des versions personnalisées et les enregistrer dans votre `IDE` pour des constructions plus rapides.

Exemples:

```
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
```

Lire Créer et utiliser plusieurs fichiers application.properties en ligne: https://riptutorial.com/fr/spring-boot/topic/6334/creer-et-utiliser-plusieurs-fichiers-application-properties

Chapitre 10: Démarrage du printemps + interface Web Hibernate + (Thymeleaf)

Introduction

Ce thread se concentre sur la façon de créer une application de démarrage à ressort avec le moteur de modèle hibernate et thymyleaf.

Remarques

Consultez également la documentation Thymeleaf

Examples

Dépendances Maven

Cet exemple est basé sur Spring Boot 1.5.1.RELEASE. avec les dépendances suivantes:

```
<!-- 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>
```

Dans cet exemple, nous allons utiliser Spring Boot JPA, Thymeleaf et les démarreurs Web.

J'utilise Lombok pour générer des getters et des setters plus faciles, mais ce n'est pas obligatoire. H2 sera utilisé comme base de données facile à configurer en mémoire.

Configuration Hibernate

Tout d'abord, permet de visualiser ce dont nous avons besoin pour configurer correctement Hibernate.

1. `@EnableTransactionManagement` et `@EnableJpaRepositories` - nous voulons une gestion transactionnelle et utiliser des référentiels de données de printemps.
2. `DataSource` - source de données principale pour l'application. en utilisant h2 en mémoire pour cet exemple.
3. `LocalContainerEntityManagerFactoryBean` - usine de gestionnaire d'entités printemps utilisant `HibernateJpaVendorAdapter` .
4. `PlatformTransactionManager` - gestionnaire de transactions principal pour les composants annotés `@Transactional` .

Fichier de configuration:

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

    @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;
    }

}
```

Entités et référentiels

Une entité simple: Utiliser les `@Getter` Lombok `@Getter` et `@Setter` pour générer des `@Getter` et des `@Setter` pour nous

```
@Entity
@Getter @Setter
public class Message {

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

}
```

J'utilise des identifiants basés sur UUID et lombok pour générer des getters et des setters.

Un référentiel simple pour l'entité ci-dessus:

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

Plus sur les dépositaires: docs de données de printemps

Assurez-vous que les entités résident dans un package mappé dans `em.setPackagesToScan` (défini dans le bean `LocalContainerEntityManagerFactoryBean`) et dans les référentiels d'un package mappé dans `basePackages` (défini dans l'annotation `@EnableJpaRepositories`)

Thymeleaf Resources et Spring Controller

Afin d'exposer les modèles Thymeleaf, nous devons définir des contrôleurs.

Exemple:

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

    @Autowired
```

```
    private MessageRepository messageRepository;

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

}
```

Ce contrôleur simple injecte `MessageRepository` et transmet tous les messages à un fichier modèle nommé `index.html` , résidant dans `src/main/resources/templates` et enfin l'expose dans `/index` .

De la même manière, nous pouvons placer d'autres modèles dans le dossier des modèles (par défaut au printemps dans `src/main/resources/templates`), leur transmettre un modèle et les transmettre au client.

Les autres ressources statiques doivent être placées dans l'un des dossiers suivants, exposés par défaut au démarrage du printemps:

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

Thymeleaf `index.html` exemple:

```
<!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` trouve dans le dossier `src/main/resources/static/css` . Vous pouvez utiliser la syntaxe `@{}` pour obtenir d'autres ressources statiques en utilisant le chemin relatif.

Lire Démarrage du printemps + interface Web Hibernate + (Thymeleaf) en ligne: https://riptutorial.com/fr/spring-boot/topic/9200/demarrage-du-printemps-plus-interface-web-

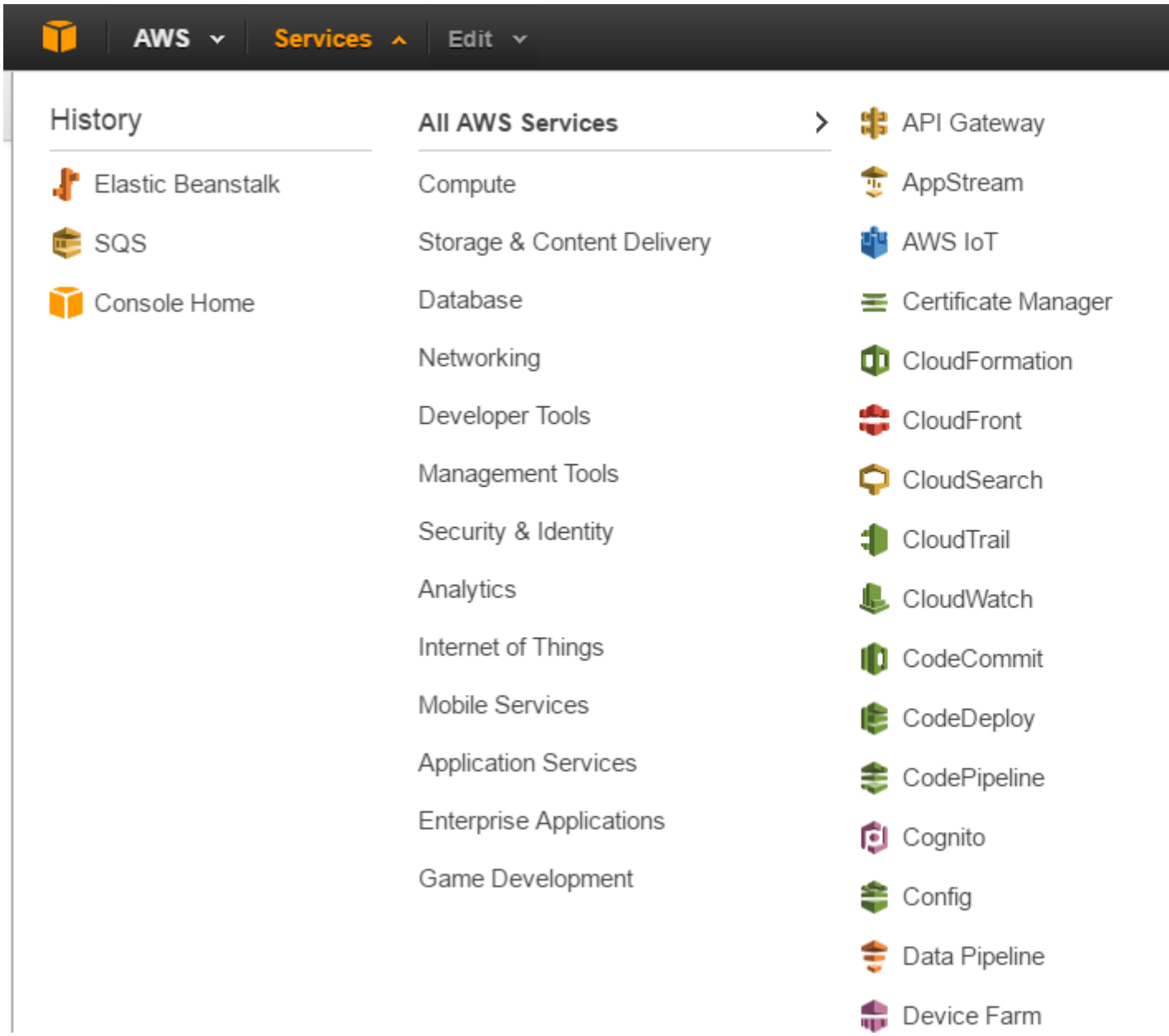
hibernate-plus--thymeleaf-

Chapitre 11: Déploiement de l'application Sample à l'aide de Spring-boot sur Amazon Elastic Beanstalk

Examples

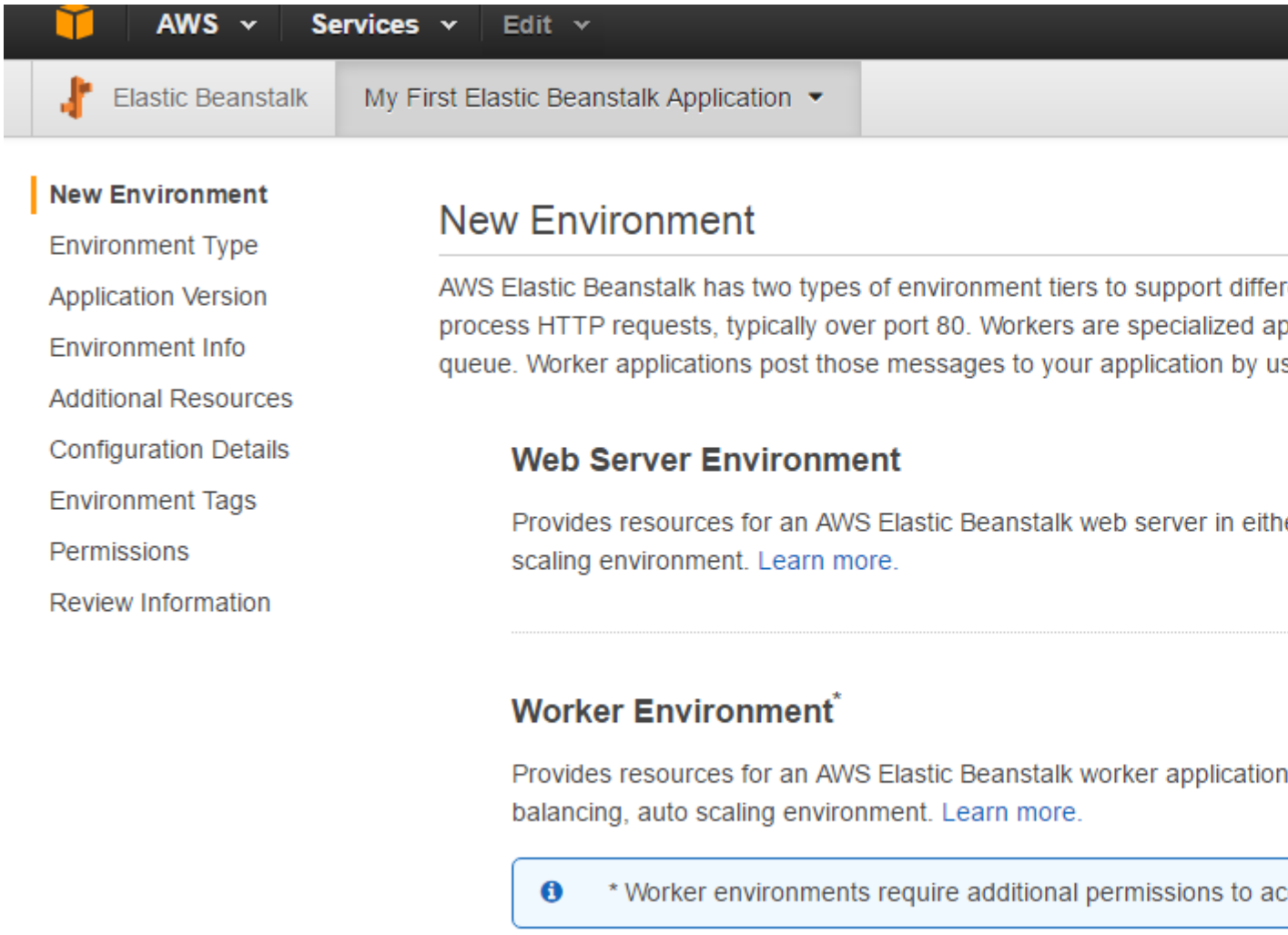
Déploiement d'un exemple d'application à l'aide de Spring-boot au format Jar sur AWS

1. Créez un exemple d'application à l'aide de Spring-Boot à partir du site de l' initialiseur Spring-Boot .
2. Importez le code dans votre IDE local et exécutez l'objectif comme **une installation propre Spring-Boot: exécutez -e**
3. Accédez au dossier cible et recherchez le fichier jar.
4. Ouvrez votre compte Amazon ou créez un nouveau compte Amazon et sélectionnez Elastic Beanstalk comme indiqué ci-dessous.



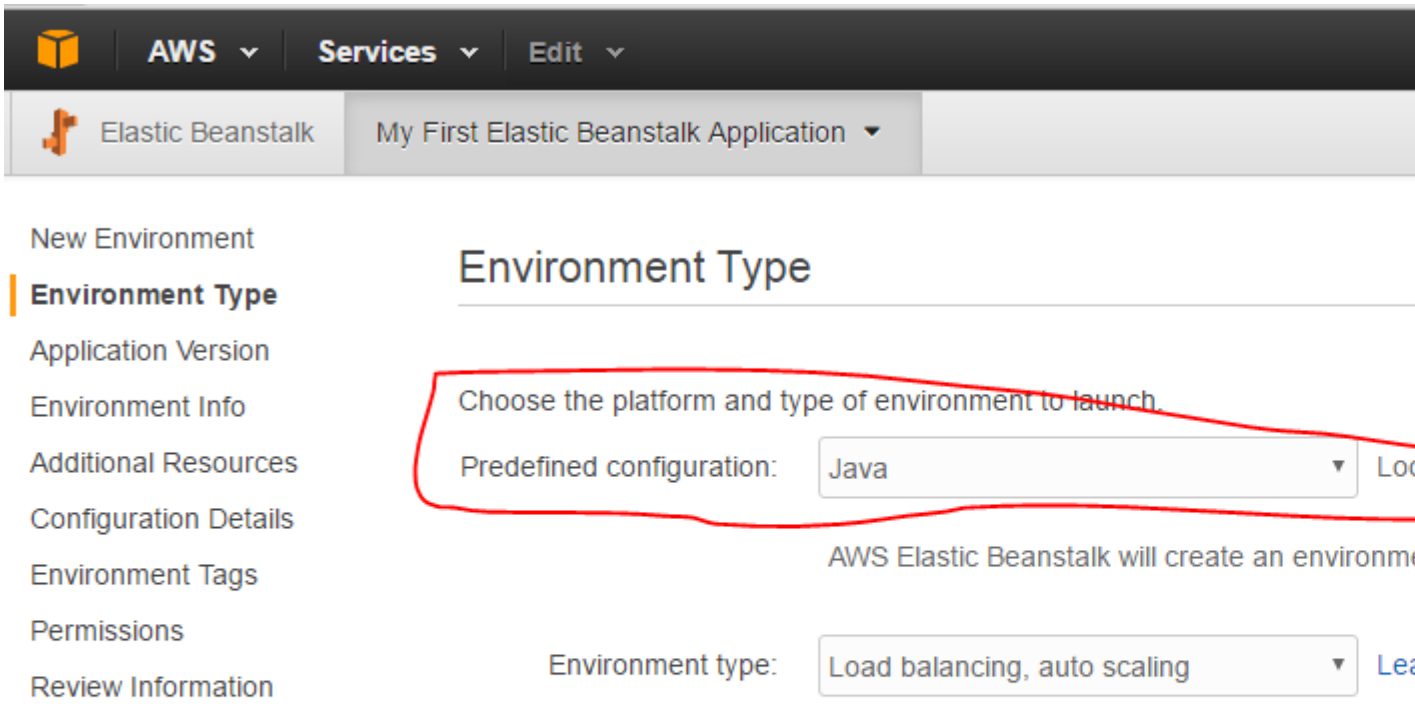
.

5. Créez un nouvel environnement de serveur Web, comme indiqué ci-dessous

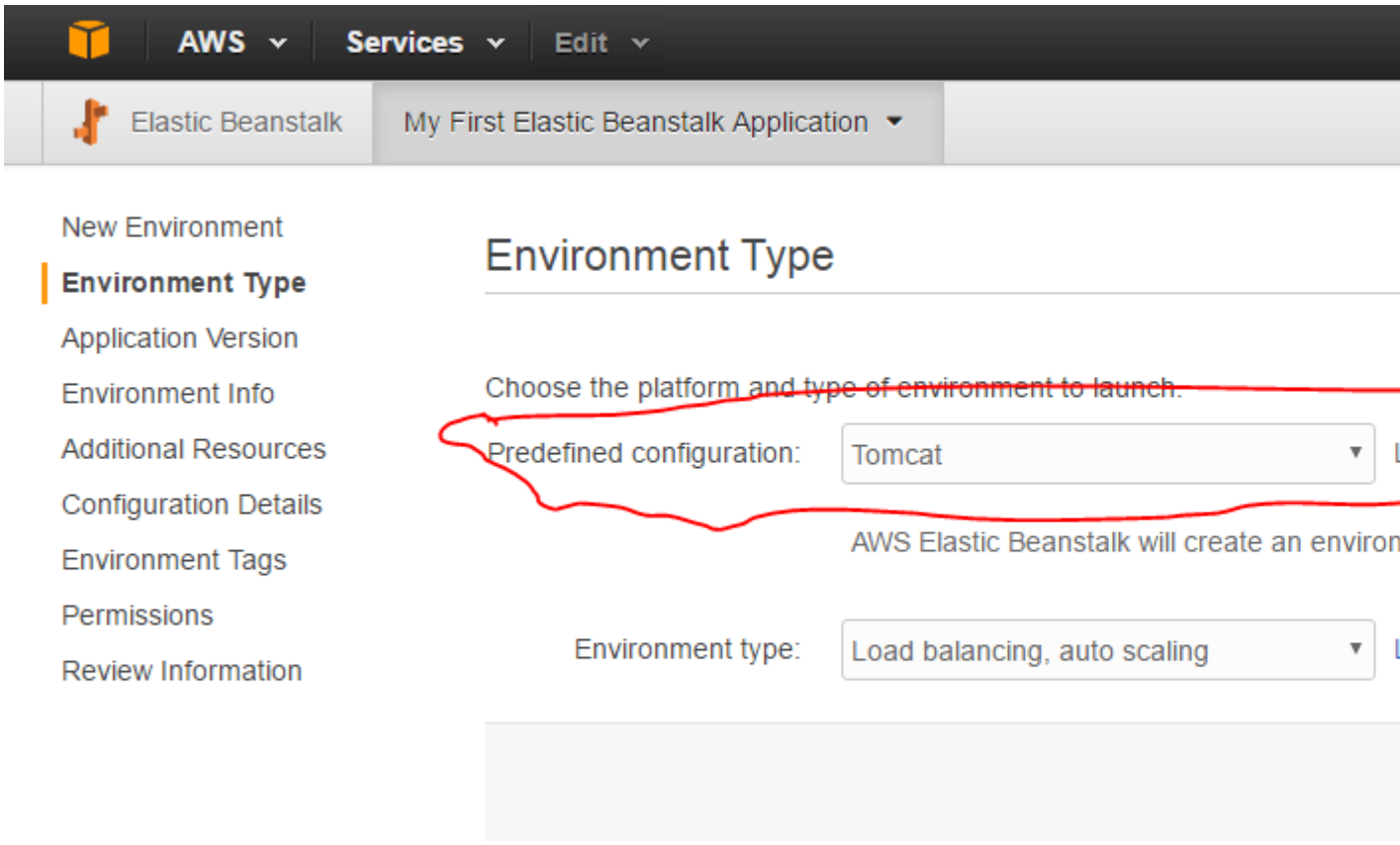


.

6. Sélectionnez le type d'environnement Java pour le déploiement de fichiers **JAR** pour l'amorçage, si vous prévoyez de le déployer en tant que fichier **WAR** , il doit être sélectionné comme tomcat, comme illustré ci-dessous.

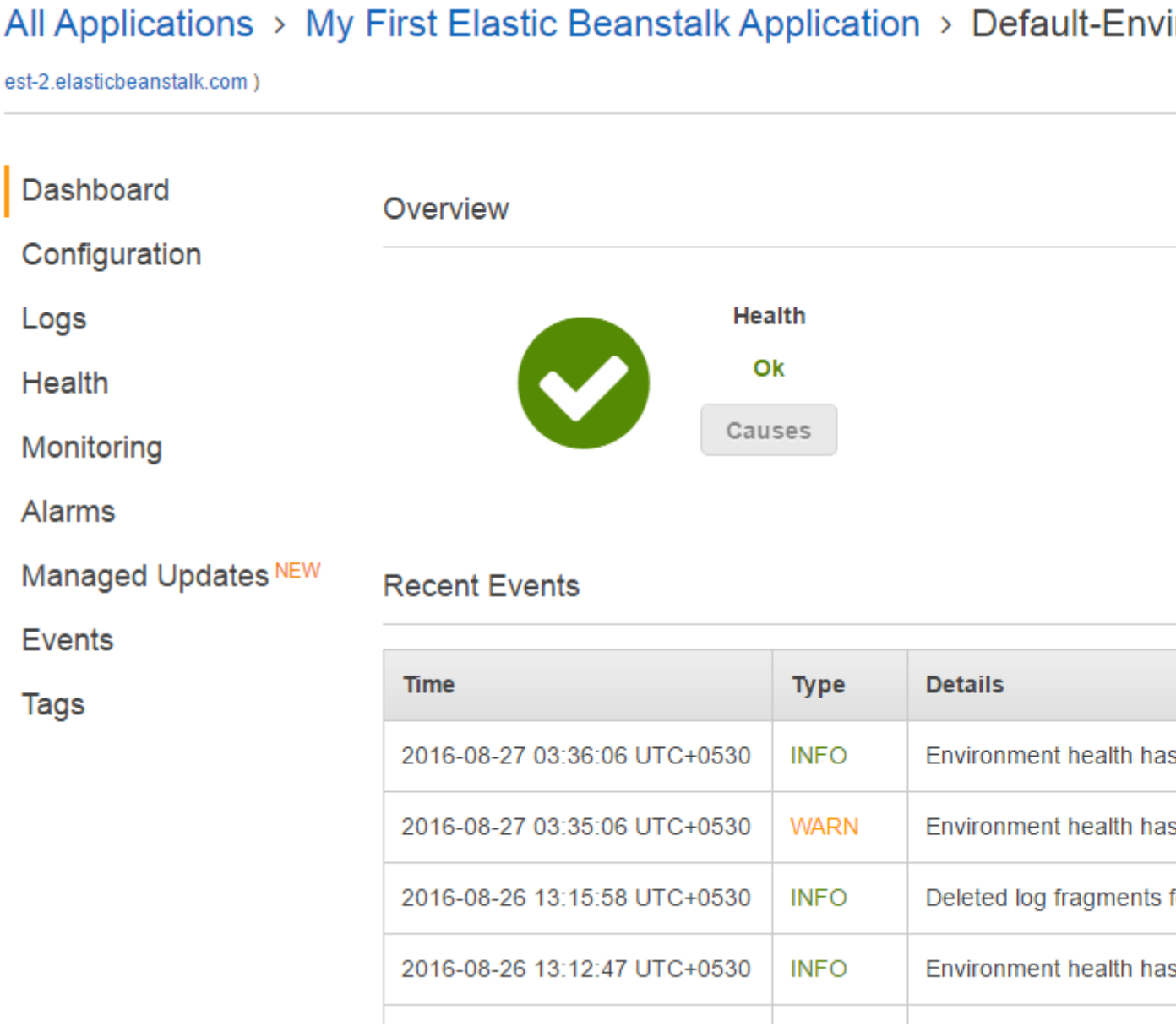

AWS
Services
Edit
Elastic Beanstalk
My First Elastic Beanstalk Application
New Environment
Environment Type
Application Version
Environment Info
Additional Resources
Configuration Details
Environment Tags
Permissions
Review Information
Environment Type
Choose the platform and type of environment to launch.
Predefined configuration:
Java
AWS Elastic Beanstalk will create an environme
Environment type:
Load balancing, auto scaling

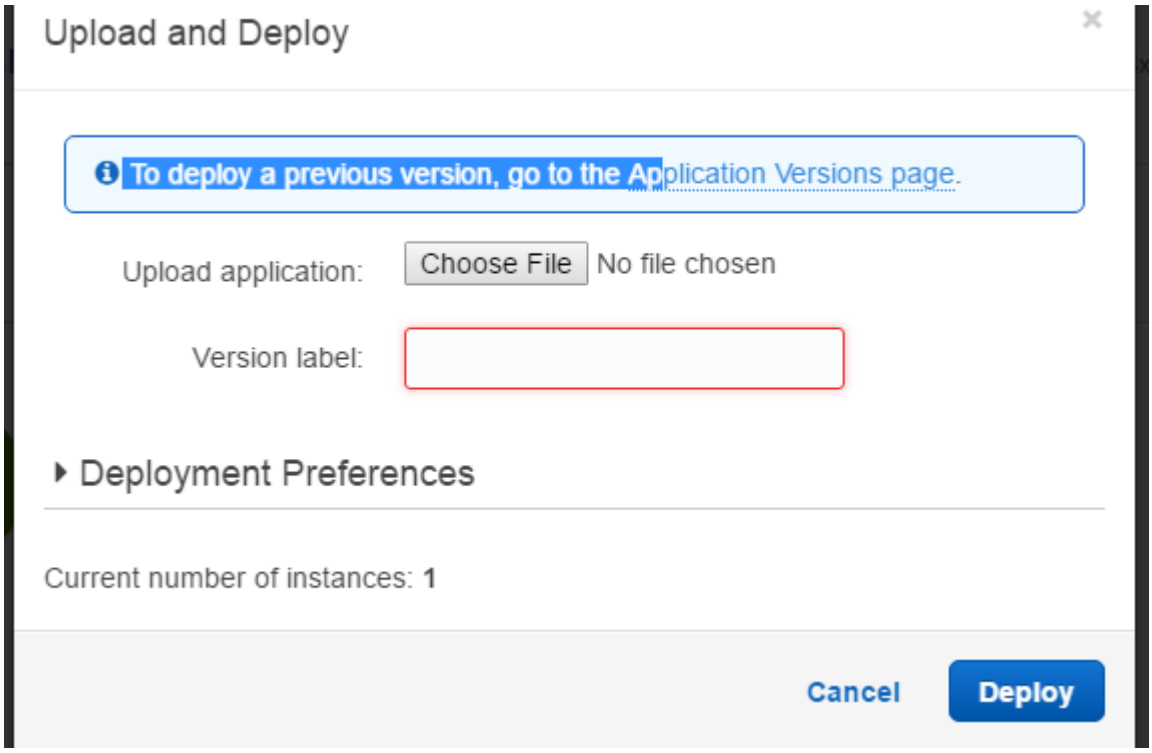
.


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
AWS Elastic Beanstalk will create an environ
Environment type:
Load balancing, auto scaling

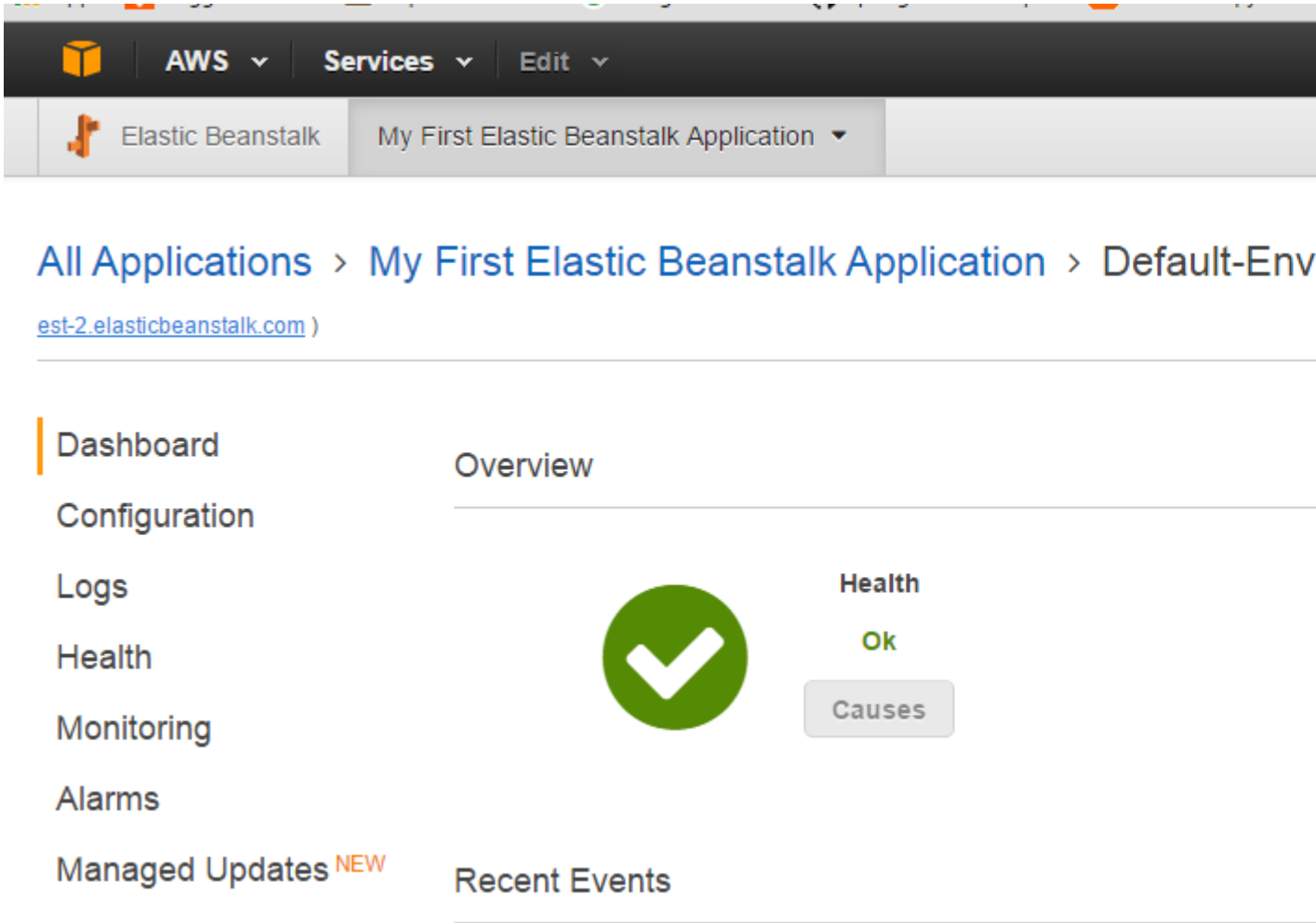
7. Sélectionnez avec la configuration par défaut en cliquant sur suivant suivant ...

8. Une fois que vous avez terminé la configuration par défaut, dans l'écran d'aperçu, le fichier JAR peut être téléchargé et déployé comme indiqué sur les figures.

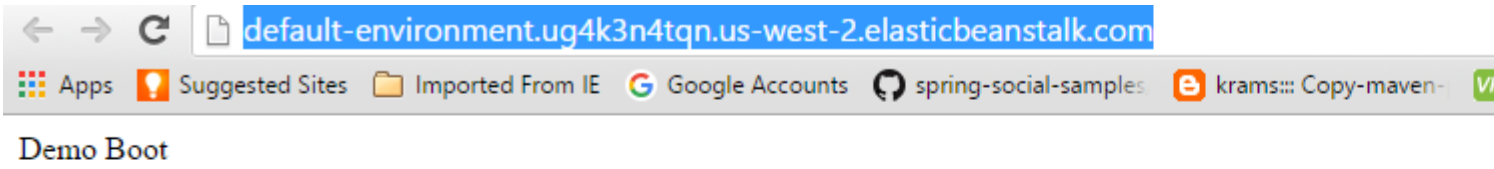




9. Une fois le déploiement réussi (5 à 10 minutes pour la première fois), vous pouvez accéder à l'URL du contexte, comme illustré ci-dessous.



10. Le résultat est comme indiqué ci-dessous, il devrait fonctionner comme avec votre env local.



11. S'il vous plaît trouver mon URL Github

Lire Déploiement de l'application Sample à l'aide de Spring-boot sur Amazon Elastic Beanstalk en ligne: https://riptutorial.com/fr/spring-boot/topic/6117/deploiement-de-l-application-sample-a-l-aide-de-spring-boot-sur-amazon-elastic-beanstalk

Chapitre 12: Installation de l'interface CLI Spring Boot

Introduction

Le CLI Spring Boot vous permet de créer et de travailler facilement avec les applications Spring Boot à partir de la ligne de commande.

Remarques

Une fois installé, l'interface CLI Spring Boot peut être exécutée à l'aide de la commande `spring` :

Pour obtenir de l'aide en ligne de commande:

```
$ spring help
```

Pour créer et exécuter votre premier projet de démarrage Spring:

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

Ouvrez votre navigateur sur `localhost:8080` :

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

Vous obtiendrez la page d'erreur whitelabel car vous n'avez pas encore ajouté de ressources à votre application, mais vous êtes prêt à utiliser uniquement les fichiers suivants:

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

- `mvnw` et `mvnw.cmd` - Des scripts d'encapsulation Maven qui téléchargent et installent Maven (si

nécessaire) lors de la première utilisation.

- `pom.xml` - La définition du projet Maven
- `DemoApplication.java` - la classe principale qui lance votre application Spring Boot.
- `application.properties` - Un fichier pour les propriétés de configuration externalisées. (Peut aussi recevoir une extension `.yml` .)
- `DemoApplicationTests.java` - Un test unitaire qui valide l'initialisation du contexte d'application Spring Boot.

Examples

Installation manuelle

Consultez la page de téléchargement pour télécharger et décompresser manuellement la dernière version, ou suivez les liens ci-dessous:

- spring-boot-cli-1.5.1.RELEASE-bin.zip
- spring-boot-cli-1.5.1.RELEASE-bin.tar.gz

Installer sur Mac OSX avec HomeBrew

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

Installer sur Mac OSX avec MacPorts

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

Installez sur n'importe quel système d'exploitation avec SDKMAN!

SDKMAN! est le gestionnaire de kit de développement logiciel pour Java. Il peut être utilisé pour installer et gérer les versions de l'interface de ligne de commande Spring Boot, ainsi que Java, Maven, Gradle, etc.

```
$ sdk install springboot
```

Lire Installation de l'interface CLI Spring Boot en ligne: https://riptutorial.com/fr/spring-boot/topic/9031/installation-de-l-interface-cli-spring-boot

Chapitre 13: Intégration Spring Boot-Hibernate-REST

Examples

Ajouter le support d'Hibernate

1. Ajoutez la dépendance **spring-boot-starter-data-jpa** à pom.xml. Vous pouvez ignorer la balise de **version** si vous utilisez **spring-boot-starter-parent** comme parent de votre **pom.xml** . La dépendance ci-dessous apporte Hibernate et tout ce qui concerne JPA à votre projet (référence).

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

2. Ajoutez le pilote de base de données à **pom.xml** . Celui-ci est pour la base de données H2 (référence).

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

3. Activer la journalisation du débogage pour Hibernate dans **application.properties**

   logging.level.org.hibernate.SQL = debug

ou dans **application.yml**

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

4. Ajoutez la classe d'entité au package souhaité sous **$ {project.home} / src / main / java /** , par exemple sous **com.example.myproject.domain** (référence):

```
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. Ajoutez **import.sql** à **$ {project.home} / src / main / resources /** . Placez les instructions **INSERT** dans le fichier. Ce fichier sera utilisé pour la population de schéma de base de données à chaque démarrage de l'application (référence):

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

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

6. Ajoutez la classe de référentiel au package souhaité sous **$ {project.home} / src / main / java /** , par exemple sous **com.example.myproject.service** (référence):

```
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);
}
```

En gros c'est ça! À ce stade, vous pouvez déjà accéder à la base de données en utilisant les méthodes de **com.example.myproject.service.CityRepository** .

Ajouter le support REST

1. Ajoutez la dépendance **spring-boot-starter-web** à pom.xml. Vous pouvez ignorer la balise de **version** si vous utilisez **spring-boot-starter-parent** comme parent de votre **pom.xml** (référence).

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

2. Ajouter le contrôleur REST au package souhaité, par exemple à **com.example.myproject.web.rest** (référence):

```
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. Démarrer l'application Spring Boot (référence).

4. Votre contrôleur est accessible à l'adresse **http: // localhost: 8080 / api / version** .

Lire Intégration Spring Boot-Hibernate-REST en ligne: https://riptutorial.com/fr/spring-boot/topic/6818/integration-spring-boot-hibernate-rest

Chapitre 14: Microservice Spring-Boot avec JPA

Examples

Classe d'application

```
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);
    }
}
```

Modèle de livre

```
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;
}
```

Juste une note puisque quelques choses se passent ici, je voulais les décomposer très rapidement.

Toutes les annotations avec @lombok génèrent une partie de la plaque chauffante de notre classe

```
@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
```

Nous avons également tiré parti de la validation dans cet objet

```
@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
```

Référentiel de livres

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

import org.springframework.data.repository.PagingAndSortingRepository;

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

Modèle de base de référentiel Spring, sauf que nous avons activé un référentiel de pagination et de tri pour des fonctionnalités supplémentaires telles que .... la pagination et le tri :)

Activation de la validation

```
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");
    }
}
```

Charger des données de test

```
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);
    }
}
```

Le simple chargement de certaines données de test devrait idéalement être ajouté uniquement dans un profil de développement.

Ajout du validateur

```
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'
```

```
}
```

Lire Microservice Spring-Boot avec JPA en ligne: https://riptutorial.com/fr/spring-boot/topic/6557/microservice-spring-boot-avec-jpa

Chapitre 15: Mise en cache avec Redis à l'aide de Spring Boot pour MongoDB

Examples

Pourquoi mettre en cache?

Aujourd'hui, la performance est l'une des mesures les plus importantes que nous devons évaluer lors du développement d'un service / d'une application Web. Garder les clients engagés est essentiel à tout produit et pour cette raison, il est extrêmement important d'améliorer les performances et de réduire les temps de chargement des pages.

Lors de l'exécution d'un serveur Web qui interagit avec une base de données, ses opérations peuvent devenir un goulot d'étranglement. MongoDB ne fait pas exception ici et, à mesure que notre base de données MongoDB évolue, les choses peuvent vraiment ralentir. Ce problème peut même s'aggraver si le serveur de base de données est détaché du serveur Web. Dans de tels systèmes, la communication avec la base de données peut entraîner une surcharge importante.

Heureusement, nous pouvons utiliser une méthode appelée mise en cache pour accélérer les choses. Dans cet exemple, nous allons présenter cette méthode et voir comment nous pouvons l'utiliser pour améliorer les performances de notre application à l'aide de Spring Cache, de Spring Data et de Redis.

Le système de base

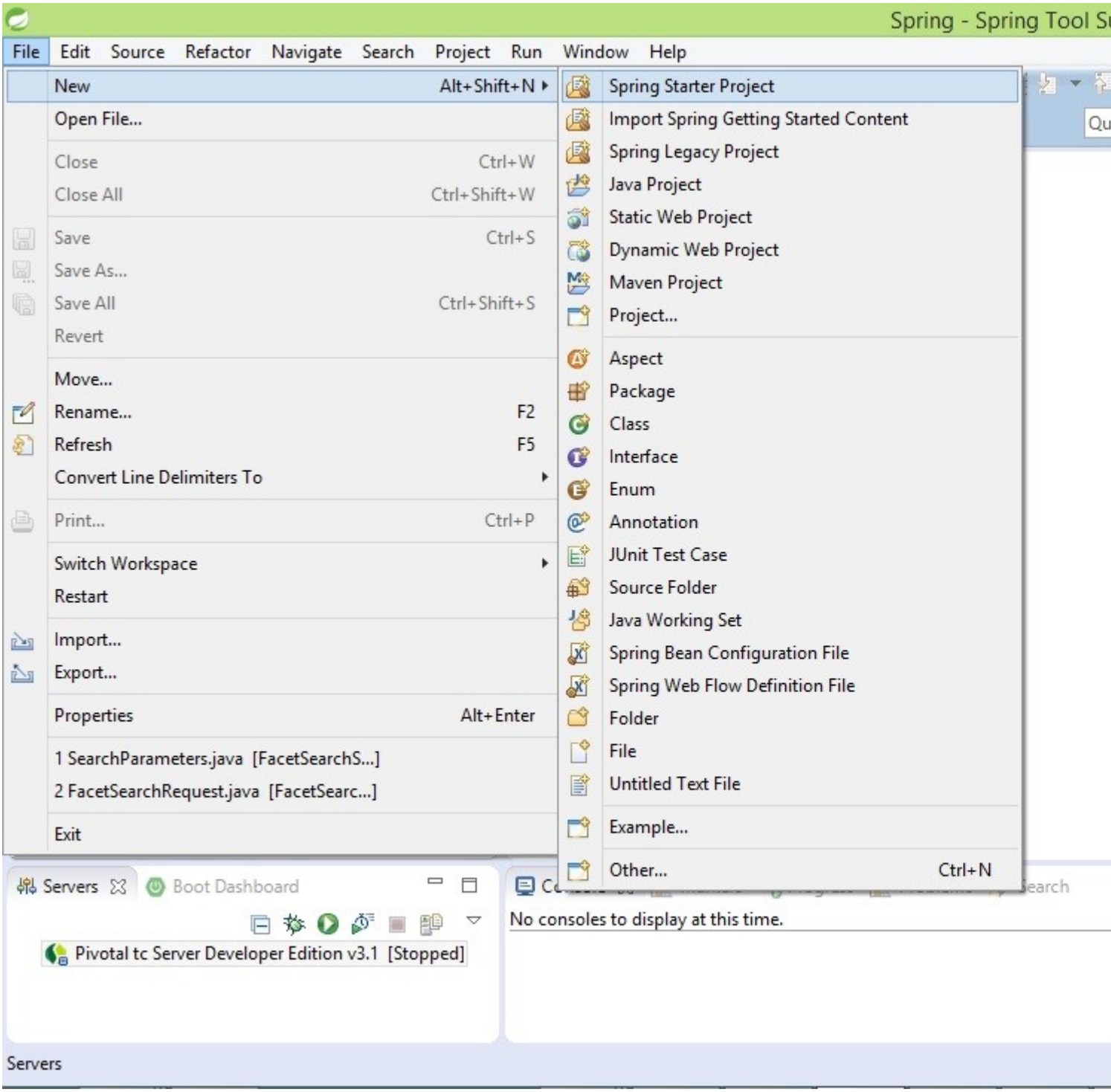
Dans un premier temps, nous allons créer un serveur Web de base qui stocke les données dans MongoDB. Pour cette démonstration, nous l'appellerons «Bibliothèque rapide». Le serveur aura deux opérations de base:

`POST /book` : cet endpoint recevra le titre, l'auteur et le contenu du livre, et créera une entrée de livre dans la base de données.

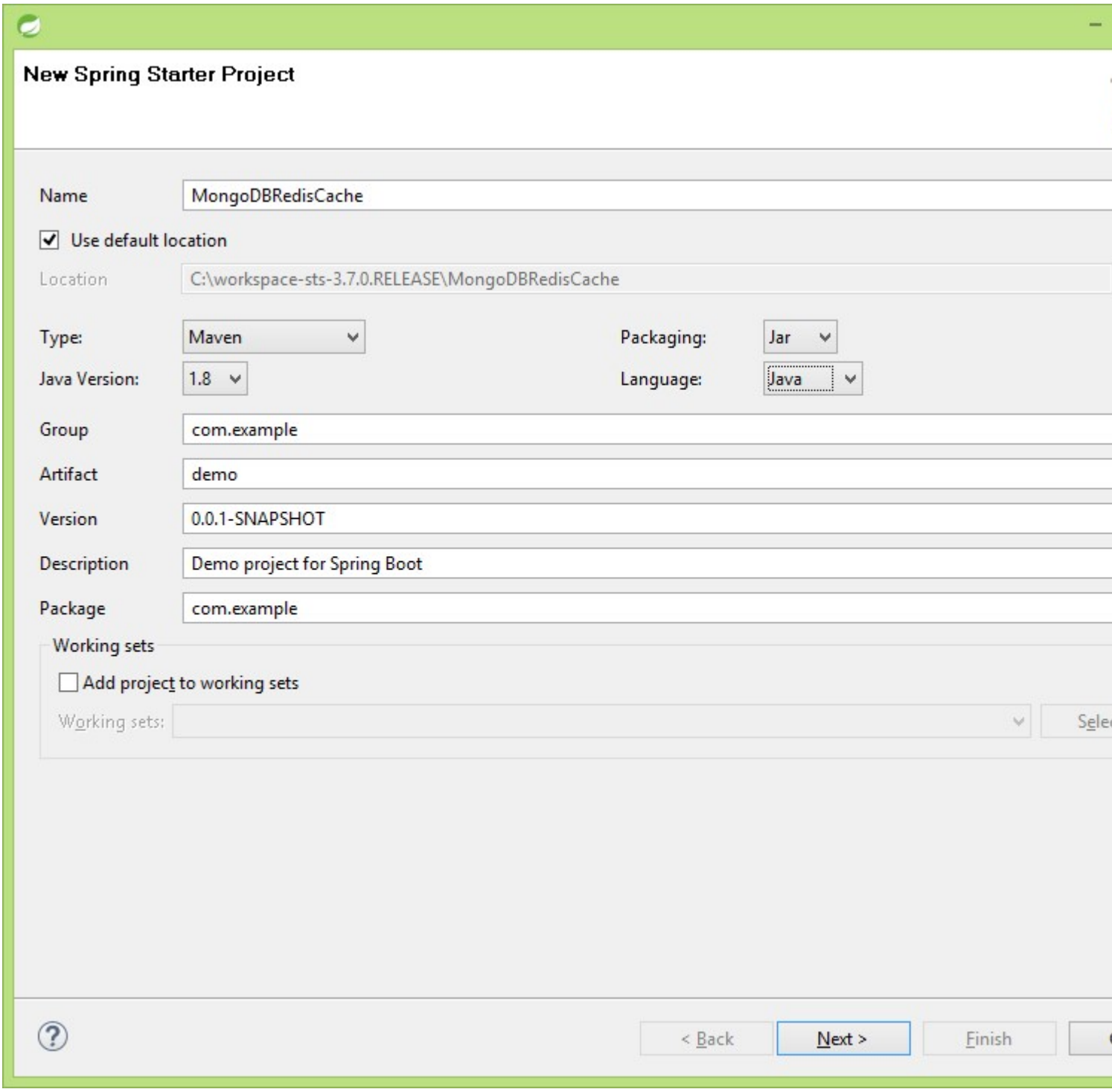
`GET /book/ {title}` : ce noeud final obtiendra un titre et retournera son contenu. Nous supposons que les titres identifient de manière unique les livres (il n'y aura donc pas deux livres avec le même titre). Une meilleure alternative serait bien sûr d'utiliser un identifiant. Cependant, pour garder les choses simples, nous utiliserons simplement le titre.

Ceci est un système de bibliothèque simple, mais nous ajouterons des capacités plus avancées ultérieurement.

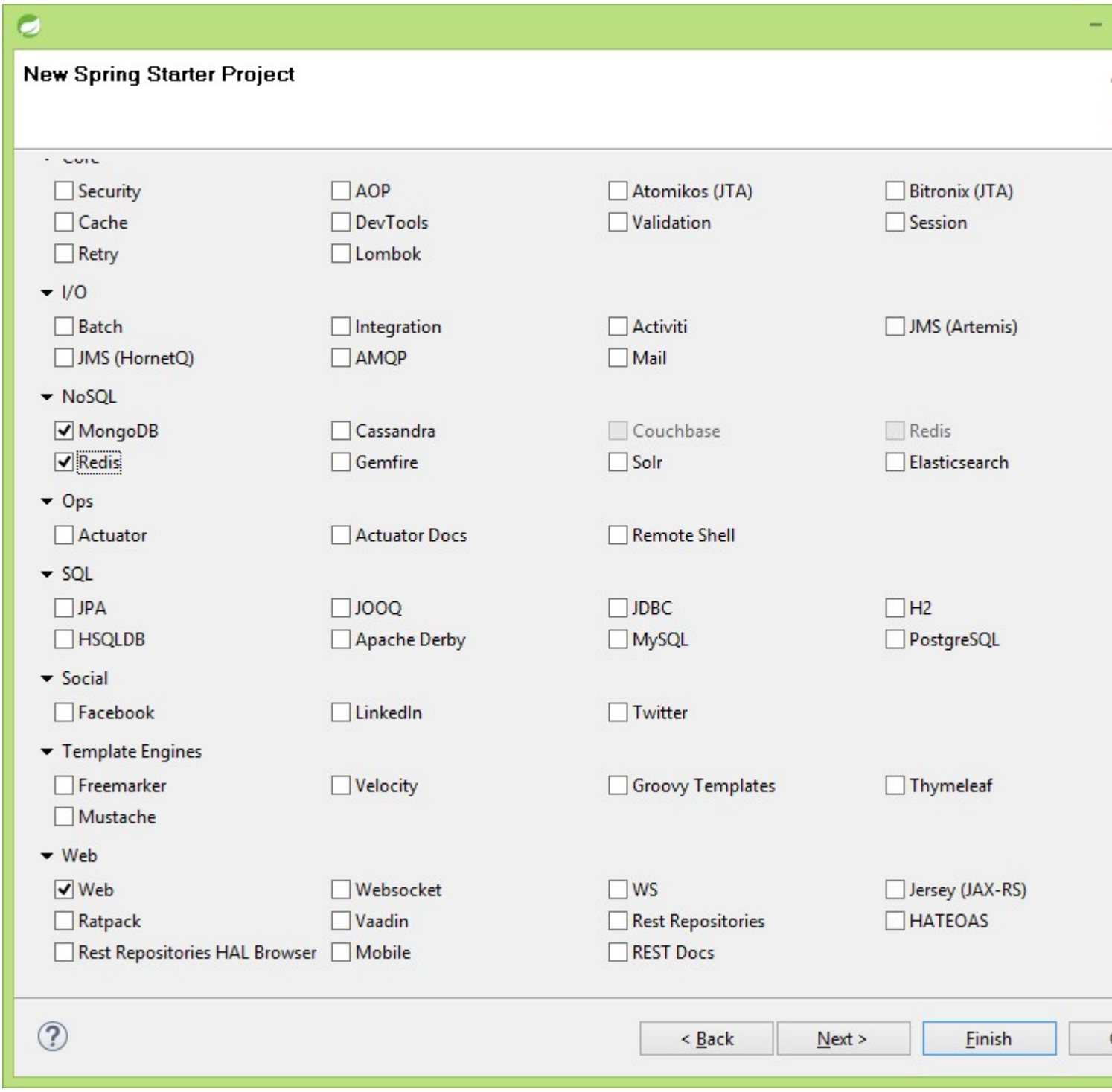
Maintenant, créons le projet en utilisant Spring Tool Suite (construction en utilisant eclipse) et Spring Starter Project



Nous construisons notre projet en utilisant Java et pour construire nous utilisons maven, sélectionnez des valeurs et cliquez sur suivant



Sélectionnez MongoDB, Redis à partir de NOSQL et Web à partir du module Web et cliquez sur Terminer. Nous utilisons Lombok pour la génération automatique des Setters et des getters de valeurs de modèle. Nous devons donc ajouter la dépendance de Lombok au POM.


New Spring Starter Project
Security
AOP
Atomikos (JTA)
Bitronix (JTA)
Cache
DevTools
Validation
Session
Retry
Lombok
I/O
Batch
Integration
Activiti
JMS (Artemis)
JMS (HornetQ)
AMQP
Mail
NoSQL
MongoDB
Cassandra
Couchbase
Redis
Redis
Gemfire
Solr
Elasticsearch
Ops
Actuator
Actuator Docs
Remote Shell
SQL
JPA
JOOQ
JDBC
H2
HSQLDB
Apache Derby
MySQL
PostgreSQL
Social
Facebook
LinkedIn
Twitter
Template Engines
Freemarker
Velocity
Groovy Templates
Thymeleaf
Mustache
Web
Web
Websocket
WS
Jersey (JAX-RS)
Ratpack
Vaadin
Rest Repositories
HATEOAS
Rest Repositories HAL Browser
Mobile
REST Docs
< Back
Next >
Finish

```
<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 contient la méthode principale utilisée pour exécuter l'application Spring Boot Application

Créer un modèle de livre contenant l'id, le titre du livre, l'auteur, la description et annoter avec @Data pour générer des règles et des getters automatiques à partir du projet 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 crée automatiquement toutes les opérations CRUD de base. Créons donc BookRepository.Java, qui recherche livre par titre et supprime le livre

```
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);
}
```

Créons webservicesController qui enregistre les données sur MongoDB et récupère les données par idTitle (titre de la propriété @PathVariable).

```
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;
    }
}
```

Ajout du cache Jusqu'à présent, nous avons créé un service Web de bibliothèque de base, mais ce n'est pas incroyablement rapide. Dans cette section, nous allons essayer d'optimiser la méthode findBookByTitle () en mettant en cache les résultats.

Pour avoir une meilleure idée de la façon dont nous allons atteindre cet objectif, revenons à l'exemple des personnes assises dans une bibliothèque traditionnelle. Disons qu'ils veulent trouver le livre avec un certain titre. Tout d'abord, ils regarderont autour de la table pour voir s'ils l'ont déjà apportée. S'ils ont, c'est génial! Ils ont juste eu un succès de cache qui trouve un élément dans le cache. S'ils ne l'ont pas trouvé, ils ont manqué le cache, ce qui signifie qu'ils n'ont pas trouvé l'élément dans le cache. Dans le cas d'un article manquant, ils devront chercher le livre dans la bibliothèque. Quand ils le trouvent, ils le gardent sur leur table ou l'insèrent dans le cache.

Dans notre exemple, nous suivrons exactement le même algorithme pour la méthode findBookByTitle (). Lorsqu'on vous demande un livre avec un certain titre, nous le rechercherons dans le cache. Si non trouvé, nous le chercherons dans le stockage principal, c'est-à-dire notre base de données MongoDB.

Utiliser Redis

L'ajout de spring-boot-data-redis à notre chemin de classe permettra au démarrage du printemps d'effectuer sa magie. Il créera toutes les opérations nécessaires en configurant automatiquement

Annotons maintenant la méthode avec la ligne ci-dessous pour mettre en cache et laisser Spring Boot faire sa magie

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

Pour supprimer du cache lorsqu'un enregistrement est supprimé, annotez simplement avec la ligne ci-dessous dans BookRepository et laissez Spring Boot gérer la suppression du cache pour nous.

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

Pour mettre à jour les données, nous devons ajouter la ligne ci-dessous à la méthode et laisser la botte de printemps gérer

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

Vous pouvez trouver le code complet du projet sur GitHub

Lire Mise en cache avec Redis à l'aide de Spring Boot pour MongoDB en ligne: https://riptutorial.com/fr/spring-boot/topic/6496/mise-en-cache-avec-redis-a-l-aide-de-spring-boot-pour-mongodb

Chapitre 16: Services REST

Paramètres

Annotation	Colonne
@Manette	Indique qu'une classe annotée est un "contrôleur" (contrôleur Web).
@RequestMapping	Annotation pour mapper des requêtes Web sur des classes de gestionnaires spécifiques (si nous les avons utilisées avec class) et / ou des méthodes de gestionnaire (si nous avons utilisé des méthodes).
method = RequestMethod.GET	Type de méthode de requête HTTP
ResponseBody	L'annotation qui indique une valeur de retour de méthode doit être liée au corps de la réponse Web
@RestController	@Controller + ResponseBody
@ResponseEntity	Extension de HttpEntity qui ajoute un code d'état HttpStatus, nous pouvons contrôler le code http de retour

Examples

Créer un service REST

1. Créez un projet en utilisant STS (Spring Starter Project) ou Spring Initializr (sur https://start.spring.io).
2. Ajoutez une dépendance Web dans votre fichier pom.xml:

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

ou tapez **Web** dans la zone `Search for dependencies` , ajoutez la dépendance Web et téléchargez le projet compressé.

3. Créer une classe de domaine (utilisateur)

```
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 getUserName() {
        return userName;
    }

    public void setUserName(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. Créez la classe UserController et ajoutez les annotations `@Controller` , `@RequestMapping`

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

5. Définir la variable statique List users pour simuler la base de données et ajouter 2 utilisateurs à la liste

```
    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. Créer une nouvelle méthode pour renvoyer tous les utilisateurs dans la liste statique (getAllUsers)

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

7. Exécutez l'application [par `mvn clean install spring-boot:run`] et appelez cette URL `http://localhost:8080/api/users`

8. Nous pouvons annoter la classe avec `@RestController` , et dans ce cas nous pouvons supprimer le ResponseBody de toutes les méthodes de cette classe `(@RestController = @Controller + ResponseBody)` , un autre point nous pouvons contrôler le code http si nous utilisons `ResponseEntity` , nous allons implémenter les mêmes fonctions précédentes mais en

utilisant `@RestController` et `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);
       }
    }
}
```

maintenant, essayez d'exécuter l'application et appelez cette URL http: // localhost: 8080 / api2 / users

Créer un service de repos avec JERSEY et Spring Boot

Jersey est l'un des nombreux frameworks disponibles pour créer des Rest Services. Cet exemple vous montre comment créer des Rest Services avec Jersey et Spring Boot.

1. Configuration du projet

Vous pouvez créer un nouveau projet en utilisant STS ou en utilisant la page Spring Initializr . Lors de la création d'un projet, incluez les dépendances suivantes:

1. Jersey (JAX-RS)
2. Web

2.Créer un contrôleur

Laissez-nous créer un contrôleur pour notre service Web Jersey

```
@Path("/Welcome")
@Component
public class MyController {
    @GET
    public String welcomeUser(@QueryParam("user") String user){
```

```
        return "Welcome "+user;
    }
}
```

`@Path("/Welcome")` indique au framework que ce contrôleur doit répondre au chemin URI / Welcome

`@QueryParam("user")` indique au framework que nous attendons un paramètre de requête avec le nom `user`

Configurations de Jersey 3.Wiring

Configurons maintenant Jersey Framework avec Spring Boot: Créez une classe, plutôt un composant spring qui étend `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")` indique au framework que seules les requêtes adressées au chemin `/MyRestService` sont censées être gérées par le framework jersey, les autres requêtes doivent toujours continuer à être gérées par le framework Spring.

C'est une bonne idée d'annoter la classe de configuration avec `@ApplicationPath` , sinon toutes les requêtes seront traitées par Jersey et nous ne pourrons pas le contourner et laisser un contrôleur de printemps le gérer si nécessaire.

4.Done

Démarrez l'application et lancez un exemple d'URL comme (en supposant que vous avez configuré le démarrage par le démarrage pour s'exécuter sur le port 8080):

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

Vous devriez voir un message dans votre navigateur comme:

Bienvenue utilisateur

Et vous avez terminé avec votre service Web Jersey avec Spring Boot

Consommer une API REST avec RestTemplate (GET)

Pour `RestTemplate` une API REST avec `RestTemplate` , créez un projet d'amorçage Spring avec initialzr d'amorçage Spring et assurez-vous que la dépendance **Web** est ajoutée:

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

Une fois votre projet configuré , créez un bean `RestTemplate` . Vous pouvez le faire dans la classe principale qui a déjà été générée ou dans une classe de configuration distincte (une classe annotée avec `@Configuration`):

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

Après cela, créez une classe de domaine, similaire à la procédure à suivre pour créer un service 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;
    }
}
```

Dans votre client, indiquez le `RestTemplate` :

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

Pour consommer une API REST qui renvoie un seul utilisateur, vous pouvez désormais utiliser:

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

En utilisant une API REST qui renvoie une liste ou un tableau d'utilisateurs, vous disposez de deux options. Soit le consomme comme un tableau:

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

Ou consommez-le en utilisant `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();
```

Sachez que lorsque vous utilisez `ParameterizedTypeReference` , vous devrez utiliser la méthode `RestTemplate.exchange()` plus avancée et vous devrez en créer une sous-classe. Dans l'exemple ci-dessus, une classe anonyme est utilisée.

Lire Services REST en ligne: https://riptutorial.com/fr/spring-boot/topic/1920/services-rest

Chapitre 17: Spring Boot + Spring Data Elasticsearch

Introduction

Spring Data Elasticsearch est une implémentation Spring Data pour Elasticsearch qui permet l'intégration au moteur de recherche Elasticsearch .

Examples

Intégration Spring Boot et Spring Data Elasticsearch

Dans cet exemple, nous allons implémenter le projet spring-data-elasticsearch pour stocker POJO dans elasticsearch. Nous verrons un exemple de projet Maven qui fait les choses suivantes:

- Insérez un élément de `Greeting(id, username, message)` sur elasticsearch.
- Obtenez tous les articles de voeux qui ont été insérés.
- Mettre à jour un élément de message d'accueil.
- Supprimer un élément de message.
- Obtenez un objet de salutation par identifiant.
- Obtenez tous les articles de vœux par nom d'utilisateur.

Intégration printanière des données d'amorçage et de données printanières

Dans cet exemple, nous allons voir une application de démarrage à ressort basée sur maven qui intègre spring-data-elasticsearch. Ici, nous allons faire les choses suivantes et voir les segments de code respectifs.

- Insérez un élément de `Greeting(id, username, message)` sur elasticsearch.
- Obtenez tous les articles de elasticsearch
- Mettre à jour un élément spécifique.
- Supprimer un élément spécifique.
- Obtenez un article spécifique par identifiant.
- Obtenez un article spécifique par nom d'utilisateur.

Fichier de configuration du projet (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>
```

Nous utiliserons Spring Boot de la version `1.5.6.RELEASE` et Spring Data Elasticsearch de cette version respective. Pour ce projet, nous devons exécuter elasticsearch-2.4.5 pour tester nos apis.

Fichier de propriétés

Nous allons mettre le fichier de propriétés du projet (nommé `applications.properties`) dans le dossier `resources` qui contient:

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

Nous utiliserons le nom, l'hôte et le port du cluster par défaut. Par défaut, le port `9300` est utilisé comme port de transport et le port `9200` est appelé port http. Pour voir le nom du cluster par défaut, cliquez sur http: // localhost: 9200 / .

Classe principale (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` est une combinaison des `@Configuration` , `@EnableAutoConfiguration` , `@EnableWebMvc` et `@ComponentScan` . La méthode `main()` utilise la méthode `SpringApplication.run()` Spring Boot pour lancer une application. Là, nous n'avons besoin d'aucune configuration xml, cette application est une pure application java spring.

Classe de configuration 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
```

```
InetSocketTransportAddress(InetAddress.getByName(EsHost), EsPort));
    }

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

`ElasticsearchConfig` classe `ElasticsearchConfig` configure elasticsearch pour ce projet et établit une connexion avec elasticsearch. Ici, `@PropertySource` est utilisé pour lire le fichier `application.properties` où sont `@PropertySource` le nom du cluster, l'hôte elasticsearch et le port. `@EnableElasticsearchRepositories` permet d'activer les référentiels Elasticsearch qui analyseront par défaut les packages de la classe de configuration annotée pour les référentiels Spring Data. `@Value` est utilisé ici pour lire les propriétés du fichier `application.properties` .

La méthode `Client()` crée une connexion de transport avec elasticsearch. La configuration ci-dessus configure un serveur Elasticsearch intégré qui est utilisé par `ElasticsearchTemplate` . Le bean `ElasticsearchTemplate` utilise le `Elasticsearch Client` et fournit une couche personnalisée pour la manipulation des données dans Elasticsearch.

Classe de modèle (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;
    }
}
```

Ici, nous avons annoté nos objets de données d' `Greeting` avec une annotation @Document que nous pouvons également utiliser pour déterminer les paramètres d'index tels que le nom, le nombre de fragments ou le nombre de répliques. L'un des attributs de la classe doit être un `id` , soit en l'annotant avec `@Id` soit en utilisant l'un des `id` ou `documentId` trouvés automatiquement. Ici, la valeur du champ `id` sera automatiquement générée si nous ne définissons aucune valeur du champ `id` .

Classe de référentiel 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);
}
```

Ici, nous avons étendu `ElasticsearchRepository` qui nous fournit de nombreux apis que nous n'avons pas besoin de définir en externe. Ceci est la classe de référentiel de base pour les classes de domaine basées sur `elasticsearch` . Comme il étend les classes de référentiels basées sur `Spring` , nous avons l'avantage d'éviter le code passe-partout requis pour implémenter les couches d'accès aux données pour différents magasins de persistance.

Ici, nous avons déclaré une méthode `findByUsername(String username)` qui convertira en une requête de correspondance qui correspond au nom d'utilisateur avec le champ `username` des objets de `Greeting` d' `Greeting` et renvoie la liste des résultats.

Services (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);
}
```

Bean de service (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 persitedGreeting = repository.findOne(greeting.getId());
        if(persitedGreeting == 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);
    }
}
```

Dans la classe ci-dessus, nous avons `@Autowired` le `GreetingRepository` . Nous pouvons simplement appeler les méthodes `CRUDRepository` et la méthode que nous avons déclarée dans la classe de référentiel avec l'objet `GreetingRepository` .

Dans la méthode `getAll()` , vous pouvez trouver une ligne `Lists.newArrayList(repository.findAll())` . Nous l'avons fait pour convertir `repository.findAll()` en

élément `List<>` car il retourne une liste `Iterable` .

Classe de contrôleur (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 idd) {
        greetingService.delete(idd);
        return new ResponseEntity<Greeting>(HttpStatus.NO_CONTENT);
    }

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

```
    }
}
```

Construire

Pour construire cette application maven exécutée

```
mvn clean install
```

Au-dessus de la commande, commencez par supprimer tous les fichiers du dossier `target` et créez le projet. Après la construction du projet, nous obtiendrons le fichier exécutable .jar nommé `springdataes-1.0-SNAPSHOT.jar` . Nous pouvons exécuter la classe principale (`Application.java`) pour démarrer le processus ou simplement exécuter le fichier jar ci-dessus en tapant:

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

Vérification des API

Pour insérer un élément de message d'accueil dans elasticsearch, exécutez la commande ci-dessous

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

Vous devriez obtenir le résultat ci-dessous comme

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

Vous pouvez également vérifier l'api get en exécutant:

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

Tu devrais obtenir

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

Vous pouvez vérifier d'autres apis en suivant les processus ci-dessus.

Documentations Officielles:

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

Lire Spring Boot + Spring Data Elasticsearch en ligne: https://riptutorial.com/fr/spring-boot/topic/10928/spring-boot-plus-spring-data-elasticsearch

Chapitre 18: Spring-Boot + JDBC

Introduction

Spring Boot peut être utilisé pour créer et conserver une base de données relationnelle SQL. Vous pouvez choisir de vous connecter à un H2 en mémoire DataBase en utilisant Spring Boot, ou peut-être choisir de vous connecter à MySql DataBase, c'est votre choix. Si vous souhaitez effectuer des opérations CRUD sur votre base de données, vous pouvez le faire en utilisant le bean JdbcTemplate, ce bean sera automatiquement fourni par Spring Boot. Spring Boot vous aidera en fournissant une configuration automatique de certains beans couramment utilisés liés à JDBC.

Remarques

Pour commencer, dans votre sts eclipse allez à nouveau -> Spring Starter Project -> remplissez vos coordonnées Maven -> et ajoutez les dépendances suivantes:

Sous l'onglet SQL -> ajoutez JDBC + ajoutez MySql (si MySql est votre choix).

Pour Mysql, vous devrez également ajouter le connecteur Java MySql.

Dans votre fichier Spring Boot application.properties (votre fichier de configuration Spring Boot), vous devez configurer vos informations d'identification de source de données sur la base de données MySql:

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

par exemple:

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

Sous le dossier resources, ajoutez les deux fichiers suivants:

1. schema.sql -> chaque fois que vous exécutez votre application, Spring Boot exécute ce fichier, à l'intérieur duquel vous devez écrire votre schéma de base de données, définir des tables et leurs relations.

2. data.sql -> Chaque fois que vous exécutez votre application, Spring Boot exécute ce fichier, à l'intérieur duquel vous devez écrire des données qui seront insérées dans votre table en tant qu'initialisation initiale.

Spring Boot vous fournira automatiquement le bean JdbcTemplate pour que vous puissiez instantanément l'utiliser comme ceci:

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

sans aucune autre configuration.

Examples

fichier 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`)
);
```

Première application de démarrage 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(..., ..., ..., ...);
```

Lire Spring-Boot + JDBC en ligne: https://riptutorial.com/fr/spring-boot/topic/9834/spring-boot-plus-jdbc

Chapitre 19: Test en botte de printemps

Examples

Comment tester une application Spring simple

Nous avons un exemple d'application d'amorçage Spring qui stocke les données utilisateur dans MongoDB et nous utilisons les services Rest pour récupérer les données.

Il y a d'abord une classe de domaine, c'est-à-dire POJO

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

    private String name;

}
```

Un référentiel correspondant basé sur Spring Data MongoDB

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

Puis notre contrôleur utilisateur

```
@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
}
```

Et enfin notre application Spring Boot

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

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

Si, disons que John Cena, The Rock et TripleHHH étaient les trois seuls utilisateurs de la base de données, une demande à / users donnerait la réponse suivante:

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

Maintenant, pour tester le code, nous allons vérifier que l'application fonctionne

```
@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);
    }
}
```

Explication

1. Comme pour tout autre test basé sur Spring, nous avons besoin de `SpringJUnit4ClassRunner` pour `SpringJUnit4ClassRunner` un contexte d'application.
2. L'annotation `@SpringApplicationConfiguration` est similaire à l'annotation `@ContextConfiguration` en ce sens qu'elle est utilisée pour spécifier le ou les contextes d'application à utiliser dans le test. En outre, il déclenchera la logique de lecture des configurations, propriétés et autres éléments spécifiques du démarrage du ressort.
3. `@WebAppConfiguration` doit être présent pour indiquer à Spring qu'un `WebApplicationContext` doit être chargé pour le test. Il fournit également un attribut pour spécifier le chemin d'accès à la racine de l'application Web.
4. `@IntegrationTest` est utilisé pour indiquer à Spring Boot que le serveur Web intégré doit être démarré. En fournissant des paires nom-valeur séparées par des deux-points ou des équations, toutes les variables d'environnement peuvent être remplacées. Dans cet exemple, le `"server.port:0"` remplacera le paramètre de port par défaut du serveur. Normalement, le serveur commence à utiliser le numéro de port spécifié, mais la valeur 0 a une signification particulière. Lorsqu'elle est définie sur 0, elle indique à Spring Boot d'analyser les ports de l'environnement hôte et de démarrer le serveur sur un port disponible aléatoire. Cela est utile si différents services occupent différents ports sur les machines de développement et le serveur de génération qui pourrait potentiellement entrer en collision avec le port de l'application, auquel cas l'application ne démarrera pas. Deuxièmement, si nous créons plusieurs tests d'intégration avec des contextes d'application différents, ils peuvent également entrer en conflit si les tests s'exécutent simultanément.
5. Nous avons accès au contexte de l'application et nous pouvons utiliser l'auto-activation pour injecter n'importe quel bean Spring.
6. La valeur `@Value("${local.server.port}")` sera résolue au numéro de port utilisé.
7. Nous créons des entités que nous pouvons utiliser pour la validation.
8. La base de données MongoDB est effacée et réinitialisée pour chaque test afin de toujours valider par rapport à un état connu. L'ordre des tests n'étant pas défini, il est probable que le test testFetchAll () échoue s'il est exécuté après le test testDeletetripleHHH ().
9. Nous demandons au Rest Assuré d'utiliser le bon port. C'est un projet open source qui fournit une DSL Java pour tester des services reposants.

10. Les tests sont implémentés en utilisant Rest Assured. nous pouvons implémenter les tests en utilisant TestRestTemplate ou tout autre client http, mais j'utilise Rest Assured car nous pouvons écrire une documentation concise en utilisant RestDocs

Chargement de différents fichiers yaml [ou properties] ou remplacement de certaines propriétés

Lorsque nous utilisons `@SpringApplicationConfiguration` il utilisera la configuration de `application.yml` [propriétés] qui, dans certaines situations, n'est pas appropriée. Donc, pour remplacer les propriétés, nous pouvons utiliser l'annotation `@TestPropertySource` .

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

    // ...

}
```

Nous pouvons utiliser l'attribut de **propriétés** de `@TestPropertySource` pour remplacer les **propriétés** spécifiques souhaitées. Dans l' exemple ci - dessus , nous **prépondérants** propriété `spring.jpa.hibernate.ddl-auto` pour `create-drop` . Et `liquibase.enabled` à `false` .

Chargement d'un fichier yml différent

Si vous souhaitez charger un fichier **yml** différent pour le test, vous pouvez utiliser l'attribut **locations** sur `@TestPropertySource` .

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

    // ...

}
```

Options alternatives

Option 1:

Vous pouvez également charger un fichier **yml** différent en plaçant un fichier **yml** sur **`test > resource`** répertoire de **`test > resource`**

Option 2:

Utiliser l'annotation `@ActiveProfiles`

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

Vous pouvez voir que nous `@ActiveProfiles` annotation `@ActiveProfiles` et que nous passons le **nom** de **fichier** comme valeur.

Créez un fichier appelé **`application-somename.yml`** et le test chargera ce fichier.

Lire Test en botte de printemps en ligne: https://riptutorial.com/fr/spring-boot/topic/1985/test-en-botte-de-printemps

Chapitre 20: ThreadPoolTaskExecutor: configuration et utilisation

Examples

configuration de l'application

```
@Configuration
@EnableAsync
public class ApplicationConfiguration{

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

}
```

Lire ThreadPoolTaskExecutor: configuration et utilisation en ligne: https://riptutorial.com/fr/spring-boot/topic/7497/threadpooltaskexecutor--configuration-et-utilisation

Crédits

S. No	Chapitres	Contributeurs
1	Commencer avec Spring-Boot	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	Analyse de paquet	Tom
3	Application Web Spring Boot entièrement réactive avec JHipster	anataliocs, codependent
4	Botte de printemps + JPA + mongoDB	Andy Wilkinson, Matthew Fontana, Rakesh, RubberDuck, Stephen Leppik
5	Botte de printemps + JPA + REST	incomplete-co.de
6	Botte de printemps + Spring Data JPA	dunni, Johir, naXa, Sanket, Sohlowmawn, sunkuet02
7	Connecter une application Spring-Boot à MySQL	koder23, sunkuet02
8	Contrôleurs	Amit Gujarathi
9	Créer et utiliser plusieurs fichiers application.properties	Patrick
10	Démarrage du printemps + interface Web Hibernate + (Thymeleaf)	ppeterka, rajadilipkolli, Tom
11	Déploiement de l'application Sample à l'aide de Spring-boot sur Amazon Elastic Beanstalk	Praveen Kumar K S
12	Installation de l'interface CLI Spring Boot	Robert Thornton

13	Intégration Spring Boot-Hibernate-REST	Igor Bljahhin
14	Microservice Spring-Boot avec JPA	Matthew Fontana
15	Mise en cache avec Redis à l'aide de Spring Boot pour MongoDB	rajadilipkolli
16	Services REST	Andy Wilkinson, CAPS LOCK, g00glen00b, Johir, Kevin Wittek, Manish Kothari, odedia, Ronnie Wang, Safwan Hijazi, shyam, Soner
17	Spring Boot + Spring Data Elasticsearch	sunkuet02
18	Spring-Boot + JDBC	Moshe Arad
19	Test en botte de printemps	Ali Dehghani, Aman Tuladhar, rajadilipkolli, Slava Semushin
20	ThreadPoolTaskExecutor: configuration et utilisation	Rosteach