1.ProjectSetup

About

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

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

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

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

Chapter 1: Getting started with spring-boot

Remarks

This section provides an overview of what spring-boot is, and why a developer might want to use it.

It should also mention any large subjects within spring-boot, and link out to the related topics. Since the Documentation for spring-boot is new, you may need to create initial versions of those related topics.

Versions

Version	Release date
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 or Setup

Getting setup with Spring Boot for the first time is quite fast thanks to the hard work of the Spring Community.

Prerequisites:

1. Java installed
2. Java IDE Recommended not required (Intellij, Eclipse, Netbeans, etc.)

You don't need to have Maven and/or Gradle installed. The projects generated by the Spring Initializr come with a Maven Wrapper (command `mvnw`) or Gradle Wrapper (command `gradlew`).

Open your web-browser to https://start.spring.io This is a launchpad for creating new Spring Boot applications for now we will go with the bare minimum.

Feel free to switch from Maven to Gradle if that is your preferred build tool.

Search for "Web" under "Search for dependencies" and add it.

Click Generate Project!

This will download a zip file called demo Feel free to extract this file wherever you want on your computer.

If you select maven please navigate a command prompt to the base directory and issue a `mvn clean install` command

You should get a build success output:

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

C:\Users\gsd4tyk\Downloads\demo>
```

Running your application: `mvn spring-boot:run`

Now your Spring Boot application starts up. Navigate your web-browser to localhost:8080

Congrats! You just got your first Spring Boot application up and running. Now lets add a tiny bit of code so you can see it working.

So use `ctrl` + `c` to exit your current running server.

Navigate to: `src/main/java/com/example/DemoApplication.java` Update this class to have a controller

```
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);
```

Good stuff now lets build and run the project again with `mvn clean install spring-boot:run`!

Now navigate your web-browser to localhost:8080

Hello World!

Congrats! We have just completed creating a Spring Boot Application and setup our first Controller to return "Hello World!" Welcome to the world of Spring Boot!

Simple Spring Boot web application using Gradle as build system

This example assumes you have already installed Java and Gradle.

Use the following project structure:

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

`build.gradle` is your build script for Gradle build system with the following content:

```
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` is the main class of the Spring Boot web application:

```
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!";
  }
}
```

Now you can run the Spring Boot web application with

```
gradle bootRun
```

and access the published HTTP endpoint either using `curl`

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

or your browser by opening localhost:8080/hello.

Chapter 2: Caching with Redis Using Spring Boot for MongoDB

Examples

Why Caching?

Today, performance is one of the most important metrics we need to evaluate when developing a web service/Application. Keeping customers engaged is critical to any product and for this reason, it is extremely important to improve the performances and reduce page load times.

When running a web server that interacts with a database, its operations may become a bottleneck. MongoDB is no exception here, and as our MongoDB database scales up, things can really slow down. This issue can even get worse if the database server is detached from the web server. In such systems, the communication with the database can cause a big overhead.

Luckily, we can use a method called caching to speed things up. In this example, we'll introduce this method and see how we can use it to enhance the performance of our application using Spring Cache, Spring Data, and Redis.

The Basic System

As the first step, we'll build a basic web server that stores data in MongoDB. For this demonstration, we'll name it "fast Library". The server will have two basic operations:

`POST /book`: This endpoint will receive the title, the author, and the content of the book, and create a book entry in the database.

`GET /book/ {title}`: This endpoint will get a title and return its content. We assume that titles uniquely identify books (thus, there won't be two books with the same title). A better alternative would be, of course, to use an ID. However, to keep things simple, we'll simply use the title.

This is a simple library system, but we'll add more advanced abilities later.

Now, let's create the project using Spring Tool Suite (build using eclipse) and spring starter Project

---

We are building our project using Java and to build we are using maven, select values and click on next

New Spring Starter Project

Name	MongoDBRedisCache

☑ Use default location

Location   C:\workspace-sts-3.7.0.RELEASE\MongoDBRedisCache

Type:	Maven ▾	Packaging:	Jar ▾
Java Version:	1.8 ▾	Language:	Java ▾

Group	com.example
Artifact	demo
Version	0.0.1-SNAPSHOT
Description	Demo project for Spring Boot
Package	com.example

Working sets

☐ Add project to working sets

Working sets: ▾   Sele

ⓘ   < Back   Next >   Finish

Select MongoDB, Redis from NOSQL and Web from the web module and click on finish. We are using Lombok for auto generation of Setters and getters of model values so we need to add the Lombok dependency to the POM

New Spring Starter Project

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

?   < 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 contains the main method which is used to run Spring Boot Application add

Create model class Book which contains id, book title, author, description and annotate with @Data to generate automatic setters and getters from jar project 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 creates all basic CRUD Operations for us automatically so let's create BookRepository.Java which finds book by title and deletes book

```
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);
}
```

Let's create webservicesController which saves data to MongoDB and retrieve data by idTitle(@PathVariable String title).

```
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;
    }
}
```

Adding the Cache So far we've created a basic library web service, but it's not astonishingly fast at all. In this section, we'll try to optimize the findBookByTitle () method by caching the results.

To get a better idea of how we'll achieve this goal, let's go back to the example of the people sitting in a traditional library. Let's say they want to find the book with a certain title. First of all, they'll look around the table to see if they already brought it there. If they have, that's great! They just had a cache hit that is finding an item in the cache. If they haven't found it, they had a cache miss, meaning they didn't find the item in the cache. In the case of a missing item, they'll have to look for the book in the library. When they find it, they'll keep it on their table or insert it into the cache.

In our example, we'll follow exactly the same algorithm for the findBookByTitle () method. When asked for a book with a certain title, we'll look for it in the cache. If not found, we'll look for it in the main storage, that is our MongoDB database.

Using Redis

Adding spring-boot-data-redis to our class path will allow spring boot to perform its magic. It will create all necessary operations by auto configuring

Let's now annotate the method with below line to cache and let spring boot do its magic

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

To delete from the cache when a record is deleted just annotate with below line in BookRepository and let Spring Boot handle cache deletion for us.

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

To update the data we need to add below line to the method and let spring boot handle

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

You can find full Project code at GitHub

Chapter 3: Connecting a spring-boot application to MySQL

Introduction

We know that spring-boot by default runs using H2 database. In this article, we will see how to tweak the default configuration to work with MySQL database.

Remarks

As a pre-requisite, make sure that MySQL is already running on port 3306 and has your database created.

Examples

Spring-boot sample using MySQL

We will follow the official guide for spring-boot and spring-data-jpa. We will be building the application using gradle.

1. Create the gradle build file

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" }
}
```

```
targetCompatibility = 1.8

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

2. Create the customer entity

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. Create Repositories

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. Create application.properties file

```
################### 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.MySQLDialec
t hibernate.show_sql=true
hibernate.hbm2ddl.auto=update
```

5. Create the PersistenceConfig.java file

In step 5, we will be defining how the datasource will be loaded and how our
application connects to MySQL. The above snippet is the bare minimum configuration
we need to connect to MySQL. Here we provide two beans:

```
@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*** This gives us an handle over the EntityManagerFactory configurations and allows us to do customizations. It also allows us to inject the PersistenceContext in our components as below:

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

- ***DataSource*** Here we return an instance of the `DriverManagerDataSource`. It is a simple implementation of the standard JDBC DataSource interface, configuring a plain old JDBC Driver via bean properties, and returning a new Connection for every getConnection call. Note that I recommend to use this strictly for testing purposes as there are better alternatives like `BasicDataSource` available. Refer here for more details

6. Create an Application class

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

}

7. Running the application

If you are using an IDE like STS, you can simply right click your project -> Run As -> Gradle (STS) Build... In the tasks list, type bootRun and Run.

If you are using gradle on command line, you can simply run the application as follows:

```
./gradlew bootRun
```

You should see something like this:

```
== 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']
```

Chapter 4: Controllers

Introduction

In this section i will add an example for the Spring boot rest controller with Get and post request.

Examples

Spring boot rest controller.

> In this example i will show how to formulate an rest controller to get and post data to the database using JPA with the most ease and least code.

In this example we will refer to the data table named buyerRequirement .

BuyingRequirement.java

@Entity @Table(name = "BUYINGREQUIREMENTS") @NamedQueries({ @NamedQuery(name = "BuyingRequirement.findAll", query = "SELECT b FROM BuyingRequirement b")}) public class BuyingRequirement extends Domain implements Serializable { private static final long 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;
}
```

}

This is the entity class which includes the parameter referring to columns in byuingRequirement table and their getters and setters.

IBuyingRequirementsRepository.java(JPA interface)

```
@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();
```

```java
    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;
    }
}
```

It includes there method

1. Post method which post data to the database .
2. Get method which fetch all records from buyingRequirement table.
3. This is also a get method which will find the buying requirement by the person's name.

Chapter 5: Create and Use of multiple application.properties files

Examples

Dev and Prod environment using different datasources

After succesfully setup Spring-Boot application all the configuration is handled in an application.properties file. You will find the file at `src/main/resources/`.

Normally there is a need to have a database behind the application. For development its good to have a setup of `dev` and a `prod` environments. Using multiple `application.properties` files you can tell Spring-Boot with which environment the application should start.

A good example is to configure two databases. One for `dev` and one for `productive`.

For the `dev` environment you can use an in-memory database like `H2`. Create a new file in `src/main/resources/` directory named `application-dev.properties`. Inside the file there is the configuration of the in-memory database:

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

For the `prod` environment we will connect to a "real" database for example `postgreSQL`. Create a new file in `src/main/resources/` directory named `application-prod.properties`. Inside the file there is the configuration of the `postgreSQL` database:

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

In your default `application.properties` file you are now able to set which profile is activated and used by Spring-Boot. Just set one attribute inside:

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

or

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

Important is that the part after `-` in `application-dev.properties` is the identifier of the file.

Now you are able to start Spring-Boot application in develop or production mode by just changing the identifier. An in-Memory database will startup or the connection to a "real" database. Sure

---

there are also much more use cases to have multiple property files.

Set the right spring-profile by building the application automatically (maven)

By creating multiple properties files for the different environments or use cases, its sometimes hard to manually change the `active.profile` value to the right one. But there is a way to set the `active.profile` in the `application.properties` file while building the application by using `maven-profiles`.

Let's say there are three environments property files in our 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
```

Those three files just differ in port and active profile name.

In the main `application.properties` file we set our spring profile using a maven variable:

application.properties.

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

After that we just have to add the maven profiles in our `pom.xml` We will set profiles for all three environments:

```
<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>
```

You are now able to build the application with maven. If you dont set any maven profile, its building the default one (in this example it's dev). For specify one you have to use a maven keyword. The keyword to set a profile in maven is `-P` directly followed by the name of the profile: `mvn clean install -Ptest`.

Now, you are also able to create custom builds and save those in your `IDE` for faster builds.

Examples:

`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
```

Chapter 6: Deploying Sample application using Spring-boot on Amazon Elastic Beanstalk

Examples

Deploying sample application using Spring-boot in Jar format on AWS

1. Create a sample application using spring-boot from spring-boot initializer site.

2. Import the code in your local IDE and run the goal as **clean install spring-boot:run -e**

3. Go to target folder and check for the jar file.

4. Open your Amazon account or create a new Amazon Account and select for the Elastic Beanstalk as shown in the below

.

5. Create a new web server environment as shown in below

New Environment

AWS Elastic Beanstalk has two types of environment tiers to support differ
process HTTP requests, typically over port 80. Workers are specialized ap
queue. Worker applications post those messages to your application by us

Web Server Environment

Provides resources for an AWS Elastic Beanstalk web server in eithe
scaling environment. Learn more.

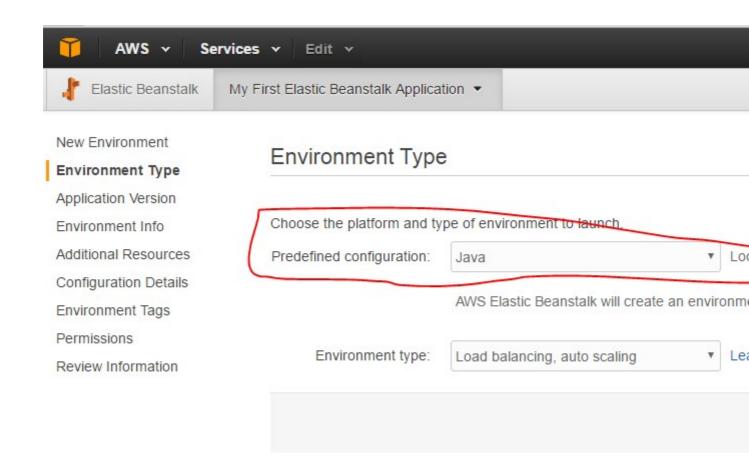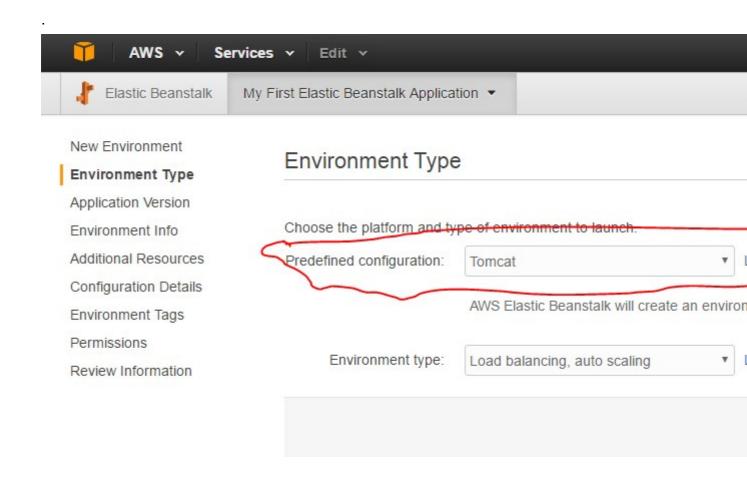### Worker Environment[*]

Provides resources for an AWS Elastic Beanstalk worker application
balancing, auto scaling environment. Learn more.

ⓘ   * Worker environments require additional permissions to ac

.

6. Select the Environment type as Java for **JAR** file deployment for Spring-boot, if you are
planning to deploy as a **WAR** file, it should be selected as tomcat as shown in below
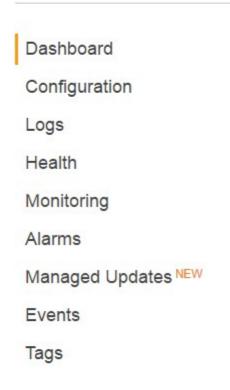
AWS ∨ **Services** ∨ Edit ∨

🐝 Elastic Beanstalk   My First Elastic Beanstalk Application ▾

New Environment
Environment Type
Application Version
Environment Info
Additional Resources
Configuration Details
Environment Tags
Permissions
Review Information

Environment Type

Choose the platform and type of environment to launch.

Predefined configuration: | Java   ▾ | Loc

AWS Elastic Beanstalk will create an environme

Environment type: | Load balancing, auto scaling   ▾ | Lea

AWS ∨ **Services** ∨ Edit ∨

🐝 Elastic Beanstalk   My First Elastic Beanstalk Application ▾

New Environment
Environment Type
Application Version
Environment Info
Additional Resources
Configuration Details
Environment Tags
Permissions
Review Information

Environment Type

Choose the platform and type of environment to launch.

Predefined configuration: | Tomcat   ▾ |

AWS Elastic Beanstalk will create an environ

Environment type: | Load balancing, auto scaling   ▾ |

---

7. Select with Default configuration's upon clicking next next ...

8. Once you complete the default configuration's, in the overview screen the JAR file can be uploaded and deployed as shown in the figures.

All Applications > My First Elastic Beanstalk Application > Default-Envi
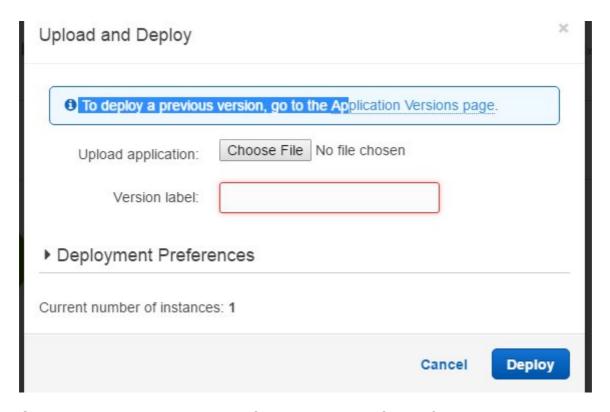
est-2.elasticbeanstalk.com)

Dashboard
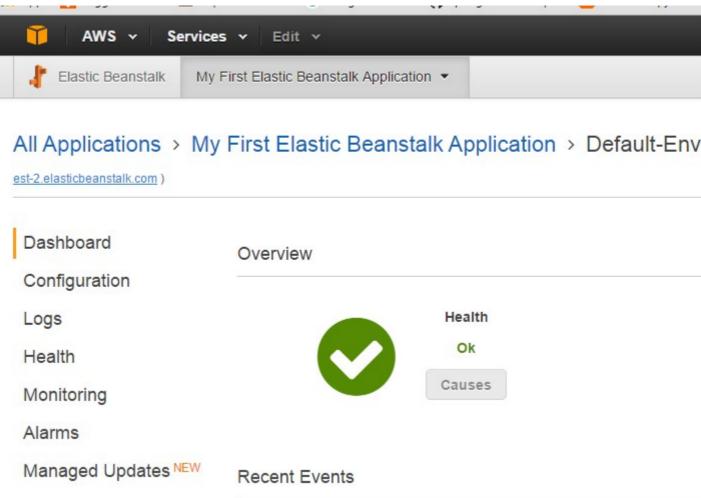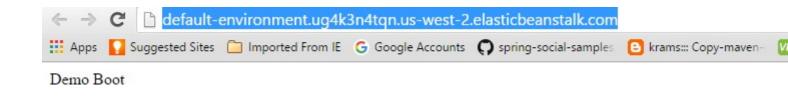
Configuration

Logs

Health

Monitoring

Alarms

Managed Updates NEW

Events

Tags

Overview

Health
Ok

Causes

Recent Events

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

Upload and Deploy   ×

ⓘ **To deploy a previous version, go to the Ap**plication Versions page.

Upload application: [Choose File] No file chosen

Version label: [   ]

▸ Deployment Preferences

Current number of instances: **1**

Cancel   **Deploy**

9. Once the Deployment is successful (5 -10 minutes for the first time) you can hit the context url as shown in the figure below.

AWS ∨ Services ∨ Edit ∨

🎵 Elastic Beanstalk   My First Elastic Beanstalk Application ▾

All Applications > My First Elastic Beanstalk Application > Default-Env

est-2.elasticbeanstalk.com)

Dashboard

Configuration

Logs

Health

Monitoring

Alarms

Managed Updates NEW

Overview

✓ Health

Ok

Causes

Recent Events

10. Result is as shown below,it should work as same as with your local env.

Demo Boot

11. Please find my Github URL

Chapter 7: Fully-Responsive Spring Boot Web Application with JHipster

Examples

Create Spring Boot App using jHipster on Mac OS

jHipster allows you to bootstrap a Spring Boot web application with a REST API back-end and a AngularJS and Twitter Bootstrap front-end.

More on jHipster here: jHipster Documentation

Install brew:

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

View additional info on how to install brew here: Install Brew

Install Gradle

Gradle is a dependency management and build system.

```
 brew install gradle
```

Install Git

Git is a version control tool

```
 brew install git
```

Install NodeJS

NodeJS gives you access to npm, the node package manager which is needed to install other tools.

```
 brew install node
```

Install Yeoman

Yeoman is a generator

```
 npm install -g yo
```

Install Bower

Bower is a dependency management tool

```
npm install -g bower
```

Install Gulp

Gulp is a task runner

```
npm install -g gulp
```

Install jHipster Yeoman Generator

This is the jHipster generator

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

Create an Application

Open a Terminal window.

Navigate to the root directory where you will keep your projects. Create an empty directory in which you will create your application

```
mkdir myapplication
```

Go to that directory

```
cd myapplication/
```

To generate your application, type

```
yo jhipster
```

You will be prompted with the following questions

Which type of application would you like to create?

Your type of application depends on whether you wish to use a microservices architecture or not. A full explanation on microservices is available here, if unsure use the default "Monolithic application".

Choose *Monolithic application* by default if you are not sure

What is your default Java package name?

Your Java application will use this as its root package.

Which type of authentication would you like to use?

Use basic session-based *Spring Security* by default if you are not sure

Which type of database would you like to use?

Which development database would you like to use?

This is the database you will use with your "development" profile. You can either use:

Use H2 by default if you are not sure

H2, running in-memory. This is the easiest way to use JHipster, but your data will be lost when you restart your server.

Do you want to use Hibernate 2nd level cache?

Hibernate is the JPA provider used by JHipster. For performance reasons, we highly recommend you to use a cache, and to tune it according to your application's needs. If you choose to do so, you can use either ehcache (local cache) or Hazelcast (distributed cache, for use in a clustered environnement)

Do you want to use a search engine in your application? Elasticsearch will be configured using Spring Data Elasticsearch. You can find more information on our Elasticsearch guide.

Choose no if you are not sure

Do you want to use clustered HTTP sessions?

By default, JHipster uses a HTTP session only for storing Spring Security's authentication and autorisations information. Of course, you can choose to put more data in your HTTP sessions. Using HTTP sessions will cause issues if you are running in a cluster, especially if you don't use a load balancer with "sticky sessions". If you want to replicate your sessions inside your cluster, choose this option to have Hazelcast configured.

Choose no if you are not sure

Do you want to use WebSockets? Websockets can be enabled using Spring Websocket. We also provide a complete sample to show you how to use the framework efficiently.

Choose no if you are not sure

Would you like to use Maven or Gradle? You can build your generated Java application either with Maven or Gradle. Maven is more stable and more mature. Gradle is more flexible, easier to extend, and more hype.

Choose *Gradle* if you are not sure

Would you like to use the LibSass stylesheet preprocessor for your CSS? Node-sass a great solution to simplify designing CSS. To be used efficiently, you will need to run a Gulp server, which will be configured automatically.

Choose no if you are not sure

Would you like to enable translation support with Angular Translate? By default JHipster provides excellent internationalization support, both on the client side with Angular Translate and on the server side. However, internationalization adds a little overhead, and is a little bit more complex to manage, so you can choose not to install this feature.

Choose no if you are not sure

Which testing frameworks would you like to use? By default JHipster provide Java unit/integration testing (using Spring's JUnit support) and JavaScript unit testing (using Karma.js). As an option, you can also add support for:

Choose none if you are not sure. You will have access to junit and Karma by default.

Chapter 8: Installing the Spring Boot CLI

Introduction

The Spring Boot CLI allows you to easily create and work with Spring Boot applications from the command-line.

Remarks

Once installed, the Spring Boot CLI can be run using the `spring` command:

To get command-line help:

```
$ spring help
```

To create and run your first Spring Boot Project:

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

Open your browser to `localhost:8080`:

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

You'll get the whitelabel error page because you haven't yet added any resources to your application, but you're all ready to go with just the following files:

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

- `mvnw` and `mvnw.cmd` - Maven wrapper scripts that will download and install Maven (if necessary) on the first use.
- `pom.xml` - The Maven project definition

- `DemoApplication.java` - the main class that launches your Spring Boot application.
- `application.properties` - A file for externalized configuration properties. (Can also be given a `.yml` extension.)
- `DemoApplicationTests.java` - A unit test that validates initialization of the Spring Boot application context.

Examples

Manual Installation

See the download page to manually download and unpack the latest version, or follow the links below:

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

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

Install on Mac OSX with HomeBrew

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

Install on Mac OSX with MacPorts

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

Install on any OS with SDKMAN!

SDKMAN! is the Software Development Kit Manager for Java. It can be used to install and manage versions of the Spring Boot CLI as well as Java, Maven, Gradle, and more.

```
$ sdk install springboot
```

Chapter 9: Package scanning

Introduction

In this topic I will overview spring boot package scanning.

You can find some basic information in spring boot docs in the following link (using-boot-structuring-your-code) but I will try to provide more detailed information.

Spring boot, and spring in general, provide a feature to automatically scan packages for certain annotations in order to create `beans` and `configuration`.

Parameters

Annotation	Details
@SpringBootApplication	Main spring boot application annotation. used one time in the application, contains a main method, and act as main package for package scanning
@SpringBootConfiguration	Indicates that a class provides Spring Boot application. Should be declared only once in the application, usually automatically by setting `@SpringBootApplication`
@EnableAutoConfiguration	Enable auto-configuration of the Spring Application Context. Should be declared only once in the application, usually automatically by setting `@SpringBootApplication`
@ComponentScan	Used to trigger automatic package scanning on a certain package and its children or to set custom package scanning
@Configuration	Used to declare one or more `@Bean` methods. Can be picked by auto package scanning in order to declare one or more `@Bean` methods instead of traditional xml configuration
@Bean	Indicates that a method produces a bean to be managed by the Spring container. Usually `@Bean` annotated methods will be placed in `@Configuration` annotated classes that will be picked by package scanning to create java configuration based beans.
@Component	By declaring a class as a `@Component` it becomes a candidates for auto-detection when using annotation-based configuration and classpath scanning. Usually a class annotated with `@Component` will become a `bean` in the application
@Repository	Originally defined by Domain-Driven Design (Evans, 2003) as

Annotation	Details
	"a mechanism for encapsulating storage. It is usualy used to indicate a `Repository` for `spring data`
@Service	Very similar in practice to `@Component`. originally defined by Domain-Driven Design (Evans, 2003) as "an operation offered as an interface that stands alone in the model, with no encapsulated state."
@Controller	Indicates that an annotated class is a "Controller" (e.g. a web controller).
@RestController	A convenience annotation that is itself annotated with `@Controller` and `@ResponseBody`. Will be automatically picked by default because it contains the `@Controller` annotation that is picked by default.

Examples

@SpringBootApplication

The most basic way to structure your code using spring boot for good automatic package scanning is using `@SpringBootApplication` annotation. This annotation provide in itself 3 other annotations that helps with automatic scanning: `@SpringBootConfiguration`, `@EnableAutoConfiguration`, `@ComponentScan` (more info about each annotation in the `Parameters` section).

`@SpringBootApplication` will usualy be placed in the main package and all other components will be placed in packages under this file:

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

Unless mentioned otherwise, spring boot detects `@Configuration`, `@Component`, `@Repository`, `@Service`, `@Controller`, `@RestController` annotations automatically under the scanned packages (`@Configuration` and `@RestController` are being picked because they are annotated by `@Component` and `@Controller` accordingly).

Basic Code example:

```
@SpringBootApplication
public class Application {

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

Setting packages/classes explicitly

Since version **1.3** you can also tell spring boot to scan specific packages by setting `scanBasePackages` or `scanBasePackageClasses` in `@SpringBootApplication` instead of specifying `@ComponentScan`.

1. `@SpringBootApplication(scanBasePackages = "com.example.myproject")` -set `com.example.myproject` as the base package to scan.
2. `@SpringBootApplication(scanBasePackageClasses = CustomerController.class)` - type-safe alternative to `scanBasePackages` sets the package of `CustomerController.java`, `com.example.myproject.web`, as the base package to scan.

Excluding auto-configuration

Another important feature is the ability to exclude specific auto-configuration classes using `exclude` or `excludeName` (`excludeName` exist since version **1.3**).

1. `@SpringBootApplication(exclude = DemoConfiguration.class)` - will exclude `DemoConfiguration` from auto package scanning.
2. `@SpringBootApplication(excludeName = "DemoConfiguration")` - will do the same using class fully classified name.

@ComponentScan

You can use `@ComponentScan` in order to configure more complex package scanning. There is also `@ComponentScans` that act as a container annotation that aggregates several `@ComponentScan` annotations.

Basic code examples

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

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

Stating `@ComponentScan` with no configuration acts like `@SpringBootApplication` and scans all packages under the class annotated with this annotation.

In this example I will state some of the useful attributes of `@ComponentScan`:

1. **basePackages** - can be used to state specific packages to scan.
2. **useDefaultFilters** - by setting this attribute to false (defaults true) you can make sure spring does not scan `@Component`, `@Repository`, `@Service`, or `@Controller` automatically.
3. **includeFilters** - can be used to *include* specific spring annotations / regex patterns to include in package scanning.
4. **excludeFilters** - can be used to *exclude* specific spring annotations / regex patterns to include in package scanning.

There are many more attributes but those are the most commonly used in order to customize package scanning.

Creating your own auto-configuration

Spring boot is based on a lot of pre-made auto-configuration parent projects. You should already be familiar with spring boot starter projects.

You can easily create your own starter project by doing the following easy steps:

1. Create some `@Configuration` classes to define default beans. You should use external properties as much as possible to allow customization and try to use auto-configuration helper annotations like `@AutoConfigureBefore`, `@AutoConfigureAfter`, `@ConditionalOnBean`, `@ConditionalOnMissingBean` etc. You can find more detailed information on each annotation in the official documentation Condition annotations
2. Place an auto-configuration file/files that aggregates all of the `@Configuration` classes.
3. Create a file named `spring.factories` and place it in `src/main/resources/META-INF`.
4. In `spring.factories`, set `org.springframework.boot.autoconfigure.EnableAutoConfiguration` property with comma separated values of your `@Configuration` classes:

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

Using this method you can create your own auto-configuration classes that will be picked by spring-boot. Spring-boot automatically scan all maven/gradle dependencies for a `spring.factories` file, if it finds one, it adds all `@Configuration` classes specified in it to its auto-configuration process.

Make sure your `auto-configuration` starter project does not contain `spring boot maven plugin` because it will package the project as an executable JAR and won't be loaded by the classpath as intended - spring boot will not be able to find your `spring.factories` and won't load your configuration

Chapter 10: REST Services

Parameters

Annotation	Column
@Controller	Indicates that an annotated class is a "Controller" (web controller).
@RequestMapping	Annotation for mapping web requests onto specific handler classes (if we used with class) and/or handler methods (if we used with methods).
method = RequestMethod.GET	Type of HTTP request methods
ResponseBody	Annotation that indicates a method return value should be bound to the web response body
@RestController	@Controller + ResponseBody
@ResponseEntity	Extension of HttpEntity that adds a HttpStatus status code, we can control the return http code

Examples

Creating a REST-Service

1. Create project using STS (Spring Starter Project) or Spring Initializr (at https://start.spring.io).
2. Add a Web Dependency in your pom.xml:

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

or type **web** in `Search for dependencies` search box, add web dependency and download zipped project.

3. Create a Domain Class (i.e. User)

```
public class User {

    private Long id;
```

```java
        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. Create UserController class and add `@Controller`, `@RequestMapping` annotations

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

5. Define static List users variable to simulate database and add 2 users to the list

```
   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. Create new method to return all users in static list (getAllUsers)

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

7. Run the application [by `mvn clean install spring-boot:run`] and call this URL
   `http://localhost:8080/api/users`

8. We can annotate the class with `@RestController`, and in this case we can remove the
   ResponseBody from all methods in this class, (`@RestController` = `@Controller` +
   `ResponseBody`), one more point we can control the return http code if we use `ResponseEntity`,
   we will implement same previous functions but using `@RestController` and `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);
      }
    }
}
```

now try to run the application and call this URL http://localhost:8080/api2/users

Creating a Rest Service with JERSEY and Spring Boot

Jersey is one of the many frameworks available to create Rest Services, This example will show you how to create Rest Services using Jersey and Spring Boot

1.Project Setup

You can create a new project using STS or by using the Spring Initializr page. While creating a project, include the following dependencies:

1. Jersey (JAX-RS)
2. Web

2.Creating a Controller

Let us create a controller for our Jersey Web Service

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

`@Path("/Welcome")` annotation indicates to the framework that this controller should respond to the URI path /Welcome

`@QueryParam("user")` annotation indicates to the framework that we are expecting one query parameter with the name `user`

3.Wiring Jersey Configurations

Let us now configure Jersey Framework with Spring Boot: Create a class, rather a spring component which extends `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")` indicates to the framework that only requests directed to the path `/MyRestService` are meant to be handled by the jersey framework, other requests should still continue to be handeld by spring framework.

It is a good idea to annotate the configuration class with `@ApplicationPath`, otherwise all the requests will be handled by Jersey and we will not be able to bypass it and let a spring controller handle it if required.

4.Done

Start the application and fire a sample URL like (Assuming you have configured spring boot to run on port 8080):

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

You should see a message in your browser like:

Welcome User

And you are done with your Jersey Web Service with Spring Boot

Consuming a REST API with RestTemplate (GET)

To consume a REST API with `RestTemplate`, create a Spring boot project with the Spring boot initialzr and make sure the **Web** dependency is added:

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

```
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Once you've set up your project, create a `RestTemplate` bean. You can do this within the main class that has already been generated, or within a separate configuration class (a class annotated with `@Configuration`):

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

After that, create a domain class, similar to how you should do when creating a REST service.

```
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;
    }
}
```

In your client, autowire the `RestTemplate`:

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

To consume a REST API that is returning a single user, you can now use:

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

Consuming a REST API that is returning a list or array of users, you have two options. Either consume it as an array:

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

Or consume it using the `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();
```

Be aware, when using `ParameterizedTypeReference`, you'll have to use the more advanced `RestTemplate.exchange()` method and you'll have to create a subclass of it. In the example above, an anonymous class is used.

Chapter 11: Spring boot + Hibernate + Web UI (Thymeleaf)

Introduction

This thread is focused on how to create a spring boot application with hibernate and thymyleaf template engine.

Remarks

Also check the Thymeleaf documentation

Examples

Maven dependencies

This example is based on spring boot 1.5.1.RELEASE. with the following dependencies:

```
<!-- 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>
```

In this example we are going to use Spring Boot JPA, Thymeleaf and web starters. I am using

Lombok to generate getters and setters easier but it is not mandatory. H2 will be used as an in-memory easy to configure database.

Hibernate Configuration

First, lets overview what we need in order to setup Hibernate correctly.

1. `@EnableTransactionManagement` and `@EnableJpaRepositories` - we want transactional management and to use spring data repositories.
2. `DataSource` - main datasource for the application. using in-memory h2 for this example.
3. `LocalContainerEntityManagerFactoryBean` - spring entity manager factory using `HibernateJpaVendorAdapter`.
4. `PlatformTransactionManager` - main transaction manager for `@Transactional` annotated components.

Configuration file:

```
@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;
    }
```

Entities and Repositories

A simple entity: Using Lombok `@Getter` and `@Setter` annotations to generate getters and setters for us

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

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

}
```

I am using UUID based ids and lombok to generate getters and setters.

A simple repository for the entity above:

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

More on respositories: spring data docs

Make sure entities reside in a package that is mapped in `em.setPackagesToScan` (defined in `LocalContainerEntityManagerFactoryBean` bean) and repositories in a package mapped in `basePackages` (defined in `@EnableJpaRepositories` annotation)

Thymeleaf Resources and Spring Controller

In order to expose Thymeleaf templates we need to define controllers.

Example:

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

    @Autowired
    private MessageRepository messageRepository;
```

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

}
```

This simple controller injects `MessageRepository` and pass all messages to a template file named `index.html`, residing in `src/main/resources/templates`, and finally expose it on `/index`.

In the same way, we can place other templates in the templates folder (default by spring to `src/main/resources/templates`), pass a model to them and serve them to the client.

Other static resources should be placed in one of the following folders, exposed by default in spring boot:

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

Thymeleaf `index.html` example:

```
<!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` is in `src/main/resources/static/css` folder. you can use the syntax `@{}` to get other static resources using relative path.

Chapter 12: Spring boot + JPA + mongoDB

Examples

CRUD operation in MongoDB using JPA

Customer Model

```
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;
    }

}
```

Customer Controller

```java
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);
    }
}
```

Customer Repository

```
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

Please add below dependencies in pom.xml file:

```
<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>
```

Insert data using rest client : POST method

For testing our application, I'm using advance rest client which is chrome extension:

So, here is the snapshot for inserting the data:

Request

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

○ GET   ● POST   ○ PUT   ○ DELETE   Other methods ▾   Custom content type ▾

Raw headers   **Headers form**

Raw payload   **Data form**

Status: 200: OK ? Loading time: 112 ms

Get Request URL

> http://localhost:8080/customer

● GET   ○ POST   ○ PUT   ○ DELETE   Other methods ▾

Raw headers   **Headers form**

Get request result:

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

Chapter 13: Spring Boot + JPA + REST

Remarks

This example uses Spring Boot, Spring Data JPA and Spring Data REST to expose a simple JPA-managed domain object via REST. The example responds with the HAL JSON format and exposes a url accessible on `/person`. The maven configuration includes a H2 in-memory database to support a quick standup.

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);
    }
}
```

Domain Object

```
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;
```

```
        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;
    }

}
```

Repository Interface

```
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) { }
```

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

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

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


}
```

Maven Configuration

```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>
    <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>
```

```
    </dependencies>
</project>
```

Chapter 14: Spring Boot + Spring Data Elasticsearch

Introduction

Spring Data Elasticsearch is a Spring Data implementation for Elasticsearch which provides integration with the Elasticsearch search engine.

Examples

Spring Boot and Spring Data Elasticsearch integration

In this example we are going to implement spring-data-elasticsearch project to store POJO in elasticsearch. We will see a sample maven project which does the followings:

- Insert a `Greeting(id, username, message)` item on elasticsearch.
- Get All Greeting items which have been inserted.
- Update a Greeting item.
- Delete a Greeting item.
- Get a Greeting item by id.
- Get all Greeting items by username.

Spring boot and spring data elasticsearch integration

In this example we are going to see a maven based spring boot application which integrates spring-data-elasticsearch. Here, we will do the followings and see the respective code segments.

- Insert a `Greeting(id, username, message)` item on elasticsearch.
- Get all items from elasticsearch
- Update a specific item.
- Delete a specific item.
- Get a specific item by id.
- Get a specific item by username.

Project configuration file (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>
```

```
    <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>
```

We will use Spring Boot of version `1.5.6.RELEASE` and Spring Data Elasticsearch of that respective version. For this project, we need to run elasticsearch-2.4.5 to test our apis.

Properties file

We will put the project properties file (named `applications.properties`) in `resources` folder which contains:

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

We will use the default cluster name, host and port. By default, `9300` port is used as transport port and `9200` port is known as http port. To see the default cluster name hit http://localhost:9200/.

Main Class(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` is a combination of `@Configuration`, `@EnableAutoConfiguration`, `@EnableWebMvc` and `@ComponentScan` annotations. The `main()` method uses Spring Boot's `SpringApplication.run()` method to launch an application. There we don't need any xml configuration, this application is pure java spring application.

Elasticsearch Configuration Class(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` class configures elasticsearch to this project and make a connection with elasticsearch. Here, `@PropertySource` is used to read the `application.properties` file where we store the cluster name, elasticsearch host and port. `@EnableElasticsearchRepositories` is used to enable Elasticsearch repositories that Will scan the packages of the annotated configuration class for Spring Data repositories by default. `@Value` is used here for reading the properties from the `application.properties` file.

The `Client()` method creates a transport connection with elasticsearch. The configuration above sets up an Embedded Elasticsearch Server which is used by the `ElasticsearchTemplate`. The `ElasticsearchTemplate` bean uses the `Elasticsearch Client` and provides a custom layer for manipulating data in Elasticsearch.

Model Class(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;
    }
}
```

Here we have annotated our `Greeting` data objects with a @Document annotation that we can also use to determine index settings like name, numbers of shards or number of replicas. One of the attributes of the class needs to be an `id`, either by annotating it with `@Id` or using one of the automatically found names `id` or `documentId`. Here, `id` field value will be auto-generated, if we don't set any value of `id` field.

Elasticsearch Repository Class(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);
}
```

Here, We have extended `ElasticsearchRepository` which provide us many of apis that we don't need to define externally. This is the base repository class for `elasticsearch` based domain classes. Since it extends `Spring` based repository classes, we get the benefit of avoiding boilerplate code required to implement data access layers for various persistence stores.

Here we have declared a method `findByUsername(String username)` which will convert to a match query that matches with username with the `username` field of `Greeting` objects and returns the list of results.

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);
}
```

Service Bean(GreetingServiceBean.java)

```
package org.springdataes.service;

import com.google.common.collect.Lists;
import org.springdataes.dao.GreetingRepository;
import org.springdataes.model.Greeting;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;


import java.util.List;

@Service
public class GreetingServiceBean implements GreetingService {

    @Autowired
    private GreetingRepository repository;

    @Override
    public List<Greeting> getAll() {
        return Lists.newArrayList(repository.findAll());
    }

    @Override
    public Greeting findOne(String id)
        { return repository.findOne(id);
    }

    @Override
    public Greeting create(Greeting greeting)
        { return repository.save(greeting);
    }

    @Override
    public Greeting update(Greeting greeting) {
        Greeting 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);
    }
}
```

In above class, we have `@Autowired` the `GreetingRepository`. We can simply call the `CRUDRepository` methods and the method we have declared in repository class with the `GreetingRepository` object.

In `getAll()` method, you may find a line `Lists.newArrayList(repository.findAll())`. We have done this to convert `repository.findAll()` to `List<>` item as it returns a `Iterable` List.

Controller Class(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);
    }
}
```

Build

To build this maven application run

```
mvn clean install
```

Above command first remove all the files in the `target` folder and build the project. After building the project we will get the executable .jar file which is named `springdataes-1.0-SNAPSHOT.jar`. We can run the main class(`Application.java`) to start the process or simply executing the above jar by typing:

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

Checking the APIs

For inserting a Greeting item in elasticsearch, execute the below command

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

You should get the below result like

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

You can also check the get api by executing:

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

You should get

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

You can check other apis by following the above processes.

Official Documentations:

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

Chapter 15: Spring boot + Spring Data JPA

Introduction

Spring Boot makes it easy to create Spring-powered, production-grade applications and services with absolute minimum fuss. It favors convention over configuration.

Spring Data JPA, part of the larger Spring Data family, makes it easy to implement JPA based repositories. It makes it easier to build apps that use data access technologies.

Remarks

Annotations

`@Repository`: Indicates that an annotated class is a "Repository", a mechanism for encapsulating storage, retrieval, and search behavior which emulates a collection of objects. Teams implementing traditional J2EE patterns such as "Data Access Object" may also apply this stereotype to DAO classes, though care should be taken to understand the distinction between Data Access Object and DDD-style repositories before doing so. This annotation is a general-purpose stereotype and individual teams may narrow their semantics and use as appropriate.

`@RestController`: A convenience annotation that is itself annotated with `@Controller` and `@ResponseBody`. Types that carry this annotation are treated as controllers where `@RequestMapping` methods assume `@ResponseBody` semantics by default.

`@Service`: Indicates that an annotated class is a "Service" (e.g. a business service facade). This annotation serves as a specialization of `@Component`, allowing for implementation classes to be autodetected through classpath scanning.

`@SpringBootApplication`: Many Spring Boot developers always have their main class annotated with `@Configuration`, `@EnableAutoConfiguration` and `@ComponentScan`. Since these annotations are so frequently used together (especially if you follow the best practices above), Spring Boot provides a convenient `@SpringBootApplication` alternative.

`@Entity`: Specifies that the class is an entity. This annotation is applied to the entity class.

Official Documentation

Pivotal Software has provided a pretty extensive documentation on Spring Framework, and it can be found at

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

Examples

Spring Boot and Spring Data JPA integration basic example

We're going to build an application that stores POJOs in a database. The application uses Spring Data JPA to store and retrieve data in a relational database. Its most compelling feature is the ability to create repository implementations automatically, at runtime, from a repository interface.

Main Class

```
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
        );
    }
}
```

The `main()` method uses Spring Boot's `SpringApplication.run()` method to launch an application. Please notice that there isn't a single line of XML. No web.xml file either. This web application is 100% pure Java and you don't have to deal with configuring any plumbing or infrastructure.

Entity Class

```
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();
```

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

Here you have a `Greeting` class with two attributes, the `id`, and the `text`. You also have two constructors. The default constructor only exists for the sake of JPA. You won't use it directly, so it can be designated as `protected`. The other constructor is the one you'll use to create instances of `Greeting` to be saved to the database.

The `Greeting` class is annotated with `@Entity`, indicating that it is a JPA entity. For lack of a `@Table` annotation, it is assumed that this entity will be mapped to a table named 'Greeting'.

The Greeting's `id` property is annotated with `@Id` so that JPA will recognize it as the object's ID. The `id` property is also annotated with `@GeneratedValue` to indicate that the ID should be generated automatically.

The other property, `text` is left unannotated. It is assumed that it'll be mapped to a column that share the same name as the property itself.

Transient Properties

In an entity class similar to the one above, we can have properties that we don't want to be persisted in the database or created as columns in our database maybe because we just want to set them at runtime and use them in our application, hence we can have that property annotated with the @Transient annotation.

```
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. */
}
```

Here you have the same Greeting class that now has a transient property `textInSomeLanguage` that can be initialized and used at runtime and won't be persisted in the database.

DAO Class

```
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` extends the `CrudRepository` interface. The type of entity and ID that it works with, `Greeting` and `Long`, are specified in the generic parameters on `CrudRepository`. By extending `CrudRepository`, `GreetingRepository` inherits several methods for working with `Greeting` persistence, including methods for saving, deleting, and finding `Greeting` entities.
See this discussion for comparison of `CrudRepository`, `PagingAndSortingRepository`, `JpaRepository`.

Spring Data JPA also allows you to define other query methods by simply declaring their method signature. In the case of `GreetingRepository`, this is shown with a `findByText()` method.

In a typical Java application, you'd expect to write a class that implements `GreetingRepository`. But that's what makes Spring Data JPA so powerful: You don't have to write an implementation of the repository interface. Spring Data JPA creates an implementation on the fly when you run the application.

Service Class

```
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
        );
    }

}
```

Controller Class

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

import java.util.Collection;
```

```java
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);
    }

}
```

Application properties file for MySQL database

```
#mysql config
spring.datasource.url=jdbc:mysql://localhost:3306/springboo
t 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.MySQLDialec
t spring.jpa.hibernate.ddl-auto = update


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


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

SQL file

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

pom.xml file

```
<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>
        <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>
```

Building an executable JAR

You can run the application from the command line with Maven. Or you can build a single executable JAR file that contains all the necessary dependencies, classes, and resources, and run that. This makes it easy to ship, version, and deploy the service as an application throughout the development lifecycle, across different environments, and so forth.

Run the application using `./mvnw spring-boot:run`. Or you can build the JAR file with `./mvnw clean package`. Then you can run the JAR file:

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

Chapter 16: Spring Boot- Hibernate-REST Integration

Examples

Add Hibernate support

1. Add **spring-boot-starter-data-jpa** dependency to pom.xml. You may skip **version** tag, if you are using **spring-boot-starter-parent** as the parent of your **pom.xml**. The dependency below brings Hibernate and everything related to JPA to your project (reference).

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

2. Add database driver to **pom.xml**. This one below is for H2 database (reference).

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

3. Enable debug logging for Hibernate in **application.properties**

logging.level.org.hibernate.SQL = debug

or in **application.yml**

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

4. Add entity class to desired package under **${project.home}/src/main/java/**, for example under **com.example.myproject.domain** (reference):

```
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. Add **import.sql** to **${project.home}/src/main/resources/**. Put **INSERT** statements into the file. This file will be used for database schema population on each start of the app (reference):

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

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

6. Add Repository class to desired package under **${project.home}/src/main/java/**, for example under **com.example.myproject.service** (reference):

```
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);
}
```

Basically that's it! At this point you already can access the database using the methods of **com.example.myproject.service.CityRepository**.

Add REST support

1. Add **spring-boot-starter-web** dependency to pom.xml. You may skip **version** tag, if you are using **spring-boot-starter-parent** as the parent of your **pom.xml** (reference).

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

2. Add REST controller to desired package, for example to **com.example.myproject.web.rest** (reference):

```
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. Start Spring Boot application (reference).

4. Your controller is accessible at the address **http://localhost:8080/api/version**.

Chapter 17: Spring-Boot + JDBC

Introduction

Spring Boot can be used to build and persist a SQL Relational DataBase. You can choose to connect to an H2 in memory DataBase using Spring Boot, or perhaps choose to connect to MySql DataBase, it's completely your choice. If you want to conduct CRUD operations against your DB you can do it using JdbcTemplate bean, this bean will automatically bean be provided by Spring Boot. Spring Boot will help you by providing auto configuration of some commonly used beans related to JDBC.

Remarks

In order to get started, in your sts eclipse go to new --> Spring Starter Project --> fill in your Maven coordinates --> and add the next dependencies:

Under SQL tab --> add JDBC + add MySql (if MySql is your choice).

For Mysql you'll also need to add the MySql Java Connector.

In you Spring Boot application.properties file (you Spring Boot configuration file) you'll need to configure your Data Source credentials to MySql DB:

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

for example:

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

Under the resources folder add the next two files:

1. schema.sql --> every time you run your application Spring Boot will run this file, inside it you suppose to write your DB schema, define tables and their relationships.

2. data.sql --> every time you run your application Spring Boot will run this file, inside it, you suppose to write data that will be inserted into your table as an initial initialization.

Spring Boot will provide you JdbcTemplate bean automatically so you can instantly can you use it like this:

```
@Autowired
```

```
private JdbcTemplate template;
```

without any other configurations.

Examples

schema.sql file

```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`)
);
```

First JdbcTemplate Boot App

```
@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(..., ..., ..., ...);
```

Chapter 18: Spring-Boot Microservice with JPA

Examples

Application Class

```
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);
    }
}
```

Book Model

```
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)
```

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

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

Just a note since a few things are going on here I wanted to break them down real quick.

All of the annotations with @lombok are generating some of our class's boiler plate

```
@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
```

We also leveraged Validation in this Object

```
@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
```

Book Repository

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

import org.springframework.data.repository.PagingAndSortingRepository;

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

Basic Spring Repository pattern, except we enabled a Paging and Sorting Repository for extra features like.…paging and sorting :)

Enabling 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");
    }
}
```

Loading some test data

```java
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);
    }
}
```

Just loading up some test data ideally this should be added only under a development profile.

Adding the Validator

```java
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'
```

Chapter 19: Testing in Spring Boot

Examples

How to Test a Simple Spring Boot Application

We have a sample Spring boot application which stores user data in MongoDB and we are using Rest services to retrieve data

First there is a domain class i.e. POJO

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

    private String name;

}
```

A corresponding repository based on Spring Data MongoDB

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

Then Our User Controller

```
@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
}
```

And finally our Spring Boot Application

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

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

If, let's say that John Cena, The Rock and TripleHHH were the only three users in the database, a request to /users would give the following response:

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

Now to test the code we will verify that the application work

```
@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);
    }
}
```

Explanation

1. Like any other Spring based test, we need the `SpringJUnit4ClassRunner` so that an application context is created.
2. The `@SpringApplicationConfiguration` annotation is similar to the `@ContextConfiguration` annotation in that it is used to specify which application context(s) that should be used in the test. Additionally, it will trigger logic for reading Spring Boot specific configurations, properties, and so on.
3. `@WebAppConfiguration` must be present in order to tell Spring that a `WebApplicationContext` should be loaded for the test. It also provides an attribute for specifying the path to the root of the web application.
4. `@IntegrationTest` is used to tell Spring Boot that the embedded web server should be started. By providing colon- or equals-separated name-value pair(s), any environment variable can be overridden. In this example, the `"server.port:0"` will override the server's default port setting. Normally, the server would start using the specified port number, but the value 0 has a special meaning. When specified as 0, it tells Spring Boot to scan the ports on the host environment and start the server on a random, available port. That is useful if we have different services occupying different ports on the development machines and the build server that could potentially collide with the application port, in which case the application will not start. Secondly, if we create multiple integration tests with different application contexts, they may also collide if the tests are running concurrently.
5. We have access to the application context and can use autowiring to inject any Spring bean.
6. The `@Value("${local.server.port}")` will be resolved to the actual port number that is used.
7. We create some entities that we can use for validation.
8. The MongoDB database is cleared and re-initialized for each test so that we always validate against a known state. Since the order of the tests is not defined, chances are that the testFetchAll() test fails if it is executed after the testDeletetripleHHH() test.
9. We instruct Rest Assured to use the correct port. It is an open source project that provides a Java DSL for testing restful services
10. Tests are implemented by using Rest Assured. we can implement the tests using the TestRestTemplate or any other http client, but I use Rest Assured because we can write concise documentation using RestDocs

Loading different yaml [or properties] file or override some properties

When we use `@SpringApplicationConfiguration` it will use configuration from `application.yml` [properties] which in certain situation is not appropriate. So to override the properties we can use `@TestPropertySource` annotation.

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

    // ...

}
```

We can use **properties** attribute of `@TestPropertySource` to override the specific **properties** we want. In above example we are **overriding** property `spring.jpa.hibernate.ddl-auto` to `create-drop`. And `liquibase.enabled` to `false`.

Loading different yml file

If you want to totally load different **yml** file for test you can use **locations** attribute on `@TestPropertySource`.

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

    // ...

}
```

Alternatively options

Option 1:

You can also load different **yml** file my placing a **yml** file on **test > resource** directory

Option 2:

Using `@ActiveProfiles` annotation

```
@RunWith(SpringJUnit4ClassRunner.class)
```

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

You can see we are using `@ActiveProfiles` annotation and we are passing the **somename** as the value.

Create a file called `application-somename.yml` and and the test will load this file.

Chapter 20: ThreadPoolTaskExecutor: configuration and usage

Examples

application configuration

```
@Configuration
@EnableAsync
public class ApplicationConfiguration{

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

}
```

Credits

S. No	Chapters	Contributors
1	Getting started with 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	Caching with Redis Using Spring Boot for MongoDB	rajadilipkolli
3	Connecting a spring-boot application to MySQL	koder23, sunkuet02
4	Controllers	Amit Gujarathi
5	Create and Use of multiple application.properties files	Patrick
6	Deploying Sample application using Spring-boot on Amazon Elastic Beanstalk	Praveen Kumar K S
7	Fully-Responsive Spring Boot Web Application with JHipster	anataliocs, codependent
8	Installing the Spring Boot CLI	Robert Thornton
9	Package scanning	Tom
10	REST Services	Andy Wilkinson, CAPS LOCK, g00glen00b, Johir, Kevin Wittek, Manish Kothari, odedia, Ronnie Wang, Safwan Hijazi, shyam, Soner
11	Spring boot + Hibernate + Web UI (Thymeleaf)	ppeterka, rajadilipkolli, Tom
12	Spring boot + JPA + mongoDB	Andy Wilkinson, Matthew Fontana, Rakesh, RubberDuck, Stephen Leppik

13	Spring Boot + JPA + REST	incomplete-co.de
14	Spring Boot + Spring Data Elasticsearch	sunkuet02
15	Spring boot + Spring Data JPA	
16	Spring Boot- Hibernate- REST Integration	dunni, Johir, naXa, Sanket, Sohlowmawn, sunkuet02
		Igor Bljahhin
17	Spring-Boot + JDBC	Moshe Arad
18	Spring-Boot Microservice with JPA	Matthew Fontana
19	Testing in Spring Boot	Ali Dehghani, Aman Tuladhar, rajadilipkolli, Slava Semushin
20	ThreadPoolTaskExecutor: configuration and usage	Rosteach