# Table of Contents

# Build RESTful APIs with Spring MVC

The Developer Notes for angularjs-springmvc-sample and angularjs-springmvc-sample-boot.

# Overview

In this minibook, I will demonstrate how to implement a RESTful web application with Spring MVC and AngularJS.

It will be consist of a series of small posts as time goes by. Every post is a standalone chapter focused on a topic.

## Assumption

I assume you are a Java developer and have some experience of Spring framework.

Else you should learn the basic Java and Java EE knowledge, and master basic usage of Spring framework.

- The official Oracle Java tutorial and Java EE tutorial are ready for Java newbies.
- Read the Spring official guides to getting started with Spring framework.

In these posts, it will not cover all Spring and Java EE features, but the following technologies will be used.

- Spring framework

  Spring framework is the infrastructure framework of this sample application.

  It provides a lightweight IOC container and a simple POJO based programming model, and also contains lots of glue codes for Java EE specification support and popular open source framework integration.

  With the benifit of Spring, it makes Java EE development without container become true, and also eases the Java EE testing. In the past years, Spring was considered as the defacto standard of Java EE development.

- Spring MVC

One of the most attractive features provided in Spring framework is the Spring MVC framework, like the old Struts framework, it is a web framework based on Servlet specification, and implements the standard MVC(Model, View, Controller) patterns.

Spring MVC supports lots of view presentations, for traditional web application or RESTful APIs. In this sample application, we only use Spring MVC as the REST API producer and exposes the APIs to client.

For the traditional web development, check my samples hosted on Spring4 sandbox.

- Spring Security

  In a traditional Java EE application, JAAS is the specification which is responsible for Authentication and Authorization. But it is very dependent on a specific container. Although most containers include a visual web UI for user maangement. But if you want to manage users and roles in program way.

  Spring Security fills this field, which makes the security control become easy, and provides a simple programming model. Spring Security is also compatible with JAAS specification, and provides JAAS integration at runtime.

  Java EE 8 is trying to introduce a new Security specification to fix this issue.

- JPA

  Based on JDBC specification, JPA provides a high level ORM abstraction and brings OOP philosophy to interact with traditional RDBMS. Hibernate and EclipseLink also support NoSQL.

- Hibernate

  In this sample application, Hibernate is used as the JPA provider. Most of time, we are trying to avoid to use a provider specific APIs, make the codes can be run cross JPA providers.

- Spring Data JPA

Spring Data JPA simplifies using JPA in Spring, including a united `Repository` to perform simple CRUD without coding, simplfied type safe Criteria Query and QueryDSL integration, a simple auditing implementation, simple pagination support of query result, Java 8 Optional and DateTime support etc.

Check the Spring Data samples in Spring4 sandbox.

We also used some third party utilities, such as Lombok project to remove the tedious getters and setters of POJOs.

For testing purpose, Spring test/JUnit, Mockito, Rest Assured will be used.

# Smaple application

In order to demonstrate how to build RESTful APIs, I will implement a simple Blog system to explain it in details.

Imagine there are two roles will use this blog system.

- ROLE_ADMIN, the administrative user.
- ROLE_USER, the normal user.

A normal user can execute the most common tasks.

1. Create a new post.
2. Update post.
3. View post detail.
4. Search posts by keyword.
5. Delete posts.
6. Comment on posts.

A administrater should have more advanced permissions, eg. he can manage the system users.

1. Create a new user.
2. Update uesr
3. Delete user
4. Search users by keyword.

# Sample codes

The complete sample codes are hosted on my Github.com account.

https://github.com/hantsy/angularjs-springmvc-sample

A Spring Boot based envolved version provides more featuers to demonstrate the cutting-edge technologies.

https://github.com/hantsy/angularjs-springmvc-sample-boot

Please read the README.md file in these respositories and run them under your local system.

# Feedback

The source of this book are hosted on my github.com account.

https://github.com/hantsy/angularjs-springmvc-sample-gitbook

Feel free to provide feedback on this project.

# An Introduction to REST

**REST** is the abbreviation of *Representational State Transter*. The term **REST** was introduced and defined in 2000 by Roy Fielding in his doctoral dissertation, Architectural Styles and the Design of Network-based Software Architectures. For Chinese users, you can find a Chinese translation copy from InfoQ.com.

# Prerequisites

Before writing any codes, please install the latest JDK 8, Apache Maven, and your favorate IDE.

## Java 8

Oracle Java 8 is recommended. For Windows user, just go to Oracle Java website to download it and install into your system. Redhat has just released a OpenJDK 8 for Windows user at DevNation 2016, if you are stick on the OpenJDK, go to Redhat Developers website and get it.

Most of the Linux distributions includes the OpenJDK, install it via the Linux package manager.

Optionally, you can set **JAVA_HOME** environment variable and add *<JDK installation dir>/bin* in your **PATH** environment variable.

Type this command in system terminal to verify your Java environment installed correctly.

```
#java -version
java version "1.8.0_102"
Java(TM) SE Runtime Environment (build 1.8.0_102-b14)
Java HotSpot(TM) 64-Bit Server VM (build 25.102-b14, mixed mode)
```

## Apache Maven

Download the latest Apache Maven from http://maven.apache.org, and uncompress it into your local system.

Optionally, you can set **M2_HOME** environment varible, and also do not forget to append *<Maven Installation dir>/bin* your **PATH** environment variable.

Type the following command to verify Apache Maven is working.

```
#mvn -v
Apache Maven 3.3.9 (bb52d8502b132ec0a5a3f4c09453c07478323dc5; 20
15-11-11T00:41:47+08:00)
Maven home: D:\build\maven
Java version: 1.8.0_102, vendor: Oracle Corporation
Java home: D:\jdk8\jre
Default locale: en_US, platform encoding: Cp1252
OS name: "windows 10", version: "10.0", arch: "amd64", family: "
dos"
```

If you are a Gradle fan, you can use Gradle as build tool. Gradle could be an alternative of Apache Maven.

## Lombok

I would like use Lombok to simply codes and make the codes clean. Go to Lombok project to get know with Lombok.

There is a good introduction to Lombok from baeldung's blog.

## IDE

The source codes are Maven based, it is IDE independent, so you can choose your favorate IDE. Nowdays the popular IDEs includes Eclipse, IDEA, NetBeans.

We will use JPA critera metadata to provide type safe query, and use Lombok to simplfy the codes, you have to enable **Annotation Processing** feature in your IDEs.

## Spring ToolSuite

Spring Tool Suite is an Eclipse based IDE, and provides a lot of built-in Spring supports, it is highly recommended for new Spring users.

Go to Spring official site, download a copy of Spring Tool Suite. At the moment, the latest version is 3.8.

Alternatively, you can download a copy of Eclipse Java EE bundle from Eclise official website, and install the STS plugin from Eclipse Marketplace.

Extract the files into your local disk. Go to root folder, there is **STS.exe** file, double click it and starts up Spring Tool Suite.

1. Go to Windows/Preference menu, and open Preference dialog
2. Search `Annotation` ...
3. Expand *Compiler/ Annotation Processing* , enable **Annotation Processing**.
4. Expand *Maven/Annotation Processing*, enable **Annotation Processing**. If it does not exists, install **m2e-apt** in **Maven/Discovery** firstly.
5. Apply all changes.

Go to Lombok project website, and follow the official the installation guideline) to install Lombok plugin into your Eclipse IDE.

## Intellij IDEA

No doubt, Intellij IDEA is the most productive Java IDE. It includes free and open source community version and enterprise version.

1. Go to File / Settings
2. Search `annotation processor`
3. Enable Annotation processing

You can install Lombok plugin from IDEA plugin manager to get Lombok support in your IDEA.

1. Go to File / Settings / Plugins
2. Click on Browse repositories...
3. Search for `Lombok Plugin`
4. Click on Install plugin
5. Restart IDE

## NetBeans

NetBeans is the simplest IDE for Java development, which was originally brought by Sun microsystem(and later maintained by Oracle), it is free and open source.

Now Oracle denoted it as an incubator project under Apache Foundation.

Download a copy of NetBeans from netbeans website(it is still working before Apache hand over it).

For NetBeans users, there is no need to setup Annotation Processing and Lombok, NetBeans has activiated Annotation processing capability by default.

In the next posts, let's try to create a project skeleton for our blog sample application.

# Getting started

Before writing codes of REST APIs for the blog sample application, we have to prepare the development environment, and create a project skeleton, and understand the basic concept and essential configurations of Spring. Then we will enrich it according to the requirements, and make it more like a real world application.

# Create project skeleton

In these days, more and more poeples are using Spring Boot to get autoconfiguration support and quicker build lifecycle.

For those new to Spring, the regular approache(none Spring Boot) is more easy to understand Spring essential configurations.

Let's us create a Maven based Java EE 7 web application to introduce how to configure a Spring MVC web application in details, then switch to Spring Boot, thus you can compare these two approaches, and understand how Spring Boot simplifies configurations.

## Create a Maven based web project

Use the official Java EE 7 web archetype to generate the project skeleton, replace the value of *archetypeId* and *package* to yours.

```
mvn -DarchetypeGroupId=org.codehaus.mojo.archetypes
-DarchetypeArtifactId=webapp-javaee7
-DarchetypeVersion=1.1
-DgroupId=your_group_id
-DartifactId=angularjs-springmvc-sample
-Dversion=1.0.0-SNAPSHOT
-Dpackage=com.hantsylabs.restsampels
-Darchetype.interactive=false
--batch-mode
archetype:generate
```

## Add Spring dependencies.

Add Spring IO platform `platform-bom` to the *dependencyManagement* section in *pom.xml*.

```
    <dependencyManagement>
        <dependencies>
            <!-- Spring BOM -->
            <dependency>
                <groupId>io.spring.platform</groupId>
                <artifactId>platform-bom</artifactId>
                <version>2.0.6.RELEASE</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
```

You can also use *platform-bom* as parent of this project.

Read the Apache Maven docs to understand Dependency Mechanism.

`platform-bom` manages all dependencies of Spring projects, and you can add dependency declaration directly without specifying a version. `platform-bom` manages the versions, and resolved potential conflicts for you.

In order to get IOC container support, you have to add the several core dependencies into *pom.xml*.

```
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-core</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-context</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-beans</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-context-support</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-aop</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-aspects</artifactId>
</dependency>
```

If you would like use `@Inject` instead of `@Autowire` in codes, add *inject* dependency.

```
<dependency>
   <groupId>javax.inject</groupId>
   <artifactId>javax.inject</artifactId>
   <version>1</version>
</dependency>
```

Get the codes from my github account to explore all configuration classes.

# Configure Spring MVC

Firstly add Spring WebMvc related dependenices into *pom.xml*.

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

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
</dependency>
```

To bootstrap a Spring MVC application, you have to enable Spring built-in `DisptachServlet`.

Serlvet 3.0 provides a new feature *ServletInitializer* to configure web applciation without *web.xml*.

Spring has its *WebApplicationInitializer* interface, there are a few classes implement this interface, `AbstractAnnotationConfigDispatcherServletInitializer` includes configuration of Spring Dispatch Servlet, and leaves some room to customize `DispatchServlet`.

Declare a `AbstractAnnotationConfigDispatcherServletInitializer` bean.

```
@Order(0)
public class AppInitializer extends AbstractAnnotationConfigDisp
atcherServletInitializer {

   @Override
   protected Class<?>[] getRootConfigClasses() {
       return new Class[] {
               AppConfig.class, //
               DataSourceConfig.class, //
```

```
                JpaConfig.class, //
                DataJpaConfig.class,//
                SecurityConfig.class,//
                Jackson2ObjectMapperConfig.class,//
                MessageSourceConfig.class
        };
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class[] {
            WebConfig.class, //
            SwaggerConfig.class //
        };
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }

    @Override
    protected Filter[] getServletFilters() {
        CharacterEncodingFilter encodingFilter = new CharacterEnc
odingFilter();
        encodingFilter.setEncoding("UTF-8");
        encodingFilter.setForceEncoding(true);

        return new Filter[] { encodingFilter };
    }

}
```

`getRootConfigClasses` specifies the configuration classes should be loaded for the Spring infrastrucuture.

`getServletConfigClasses` specifies the configurations depend on Servlet specification, esp, web mvc related configurations.

`getServletMappings` is the Spring `DispatchServlet` mapping URL pattern.

`getServletFilters` are those Web Filters will be applied on the Spring `DispatchServlet`.

Spring MVC `DispatchServlet` is configured in the super classes, explore the details if you are interested in it.

In `getServletConfigClasses`, we specify a `WebConfig` will be loaded, which is responsible for configuring Spring MVC in details, including resource handling, view, view resolvers etc.

A classic `WebConfig` looks like.

```
public class WebConfig extends WebMvcConfigurerAdapter {

}
```

Generally, `WebMvcConfigurerAdapter` is the extension point left for users to use for customizing MVC configurations.

Spring Data project provides a `SpringDataWebConfiguration` which is a subclass of `WebMvcConfigurerAdapter` and adds pagination, sort, and domain object conversion support. Open the source code of `SpringDataWebConfiguration` and research yourself.

The following is the full source code of `WebConfig`.

```
@Configuration
@EnableWebMvc
@ComponentScan(
        basePackageClasses = {Constants.class},
        useDefaultFilters = false,
        includeFilters = {
            @Filter(
                    type = FilterType.ANNOTATION,
                    value = {
                        Controller.class,
                        RestController.class,
                        ControllerAdvice.class
                    }
            )
```

```
        }
)
public class WebConfig extends SpringDataWebConfiguration {

    private static final Logger logger = LoggerFactory.getLogger
(WebConfig.class);

    @Inject
    private ObjectMapper objectMapper;

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry regi
stry) {

        registry.addResourceHandler("/swagger-ui.html")
                .addResourceLocations("classpath:META-INF/resour
ces/");

        registry.addResourceHandler("/webjars/**")
                .addResourceLocations("classpath:META-INF/resour
ces/webjars/");
    }

    @Override
    public void addViewControllers(ViewControllerRegistry regist
ry) {

    }

    @Override
    public void configureHandlerExceptionResolvers(List<HandlerE
xceptionResolver> exceptionResolvers) {
        exceptionResolvers.add(exceptionHandlerExceptionResolver
());
    }

    @Override
    public void configureDefaultServletHandling(DefaultServletHa
ndlerConfigurer configurer) {
        configurer.enable();
```

```
    }

    @Override
    public void configureContentNegotiation(ContentNegotiationConfigurer configurer) {
        configurer.favorParameter(false);
        configurer.favorPathExtension(false);
    }

    @Override
    public void configureMessageConverters(List<HttpMessageConverter<?>> converters) {
        List<HttpMessageConverter<?>> messageConverters = messageConverters();
        converters.addAll(messageConverters);
    }

    @Bean
    public ExceptionHandlerExceptionResolver exceptionHandlerExceptionResolver() {
        ExceptionHandlerExceptionResolver exceptionHandlerExceptionResolver = new ExceptionHandlerExceptionResolver();
        exceptionHandlerExceptionResolver.setMessageConverters(messageConverters());

        return exceptionHandlerExceptionResolver;
    }

    private List<HttpMessageConverter<?>> messageConverters() {
        List<HttpMessageConverter<?>> messageConverters = new ArrayList<>();

        MappingJackson2HttpMessageConverter jackson2Converter = new MappingJackson2HttpMessageConverter();
        jackson2Converter.setSupportedMediaTypes(Arrays.asList(MediaType.APPLICATION_JSON));
        jackson2Converter.setObjectMapper(objectMapper);

        messageConverters.add(jackson2Converter);
        return messageConverters;
```

```
    }

}
```

- `@EnableWebMvc` tells Spring to enable Spring MVC.
- `@ComponentScan` uses a filter to select Spring MVC related classes to be activated.
- `addResourceHandlers` configure how to handle static resources.
- `addViewControllers` leave places to configure view resolver to render specific views.
- `configureHandlerExceptionResolvers` specifies exception handling stretagy.
- `configureMessageConverters` configure `HttpMessageConverter` will be used for serialization and deserialization.

I would like use `application/json` as default content type, and uses Jackson to serialize and deserialize messages.

Configure a Jackson ObjectMapper as you need.

```
@Configuration
public class Jackson2ObjectMapperConfig {

    @Bean
    public ObjectMapper objectMapper() {

        Jackson2ObjectMapperBuilder builder = new Jackson2Object
MapperBuilder();
        builder.serializationInclusion(Include.NON_EMPTY);
        builder.featuresToDisable(
               // SerializationFeature.WRITE_DATES_AS_TIMESTAMPS,

                DeserializationFeature.FAIL_ON_IGNORED_PROPERTIE
S,
                DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIE
S);
        builder.featuresToEnable(DeserializationFeature.ACCEPT_S
INGLE_VALUE_AS_ARRAY);

        return builder.build();
    }
}
```

`Jackson2ObjectMapperBuilder` provides fluent APIs to configure which features will be enabled or disabled for serialization and deserialization. For the complete configurable options, check the javadocs of `SerializationFeature` and `DeserializationFeature` .

# Configure DataSource

In order to use Hibernate, Jdbc, or JPA similar persistence framework or tools, you have to configure a `java.sql.DataSource` for it.

Spring DataSource support is available in `sring-jdbc`. Added it into your *pom.xml*.

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
</dependency>
```

A simple `DataSouce` configuration looks like.

```
@Configuration
public class DataSourceConfig {

    @Bean
    public DataSource testDataSource() {
        BasicDataSource bds = new BasicDataSource();
        bds.setDriverClassName("com.mysql.jdbc.Driver");
        bds.setUrl("jdbc:mysql://localhost:3306");
        bds.setUsername("jdbc.username");
        bds.setPassword("jdbc.password");
        return bds;
    }

}
```

Here, I uses Apache Commons Dbcp's `BasicDataSource` to build a `DataSource`. It is configured for MySQL database, before use it, do not forget to add mysql driver into *pom.xml*.

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
```

Declares this configuration class in `getRootConfigClasses` method of `AppInitializer`.

In above codes, we set username, password etc in hard codes, but in a real application, it is better to externalize these configurations into a property file.

Create another `@configuration` class for this purpose.

```
@Configuration
@ComponentScan(
    basePackageClasses = {Constants.class},
    excludeFilters = {
        @Filter(
            type = FilterType.ANNOTATION,
            value = {
                RestController.class,
                ControllerAdvice.class,
                Configuration.class
            }
        )
    }
)
@PropertySource("classpath:/app.properties")
@PropertySource(value = "classpath:/database.properties", ignore
ResourceNotFound = true)
public class AppConfig {

}
```

`AppConfig` work as an entr configuration for this application. `@ComponentScan` use a fitler to load all none web components.

Use `@PropertySource` to load the external properties files, `app.properties` is use for application properties, and database.properties for holding database datasource properties.

```
jdbc.url=@jdbc.url@
jdbc.username=@jdbc.username@
jdbc.password=@jdbc.password@
hibernate.dialect=@hibernate.dialect@
```

In DataSouce configuration, use `Environment` to fetch these properties.

```
private static final String ENV_JDBC_PASSWORD = "jdbc.password";
private static final String ENV_JDBC_USERNAME = "jdbc.username";
private static final String ENV_JDBC_URL = "jdbc.url";

@Inject
private Environment env;

@Bean
public DataSource testDataSource() {
    BasicDataSource bds = new BasicDataSource();
    bds.setDriverClassName("com.mysql.jdbc.Driver");
    bds.setUrl(env.getProperty(ENV_JDBC_URL));
    bds.setUsername(env.getProperty(ENV_JDBC_USERNAME));
    bds.setPassword(env.getProperty(ENV_JDBC_PASSWORD));
    return bds;
}
```

Spring Jdbc provides a simple `EmbeddedDatabaseBuilder` to build an embedded datasource on the fly way.

```
@Bean
public DataSource dataSource() {
    return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.H2)
            .build();
}
```

Here we build an embedded H2 datasource.

An embedded datasource is every helpful for development stage, everytime when we run the application, or run the tests, we are getting a fresh runtime environment.

Spring Jdbc also provides other built-in DataSource, such as DriverManagerDataSource, and some application server specific DataSource, eg. for Webphere.

For a production runtime environment, we should use pooled datasource, such as Apache Commons Dbcp, or application server built-in DataSource to get better performance.

We have discussed the usages of Apache Commons Dbcp earlier, you can add extra pool configuration for this datasource.

For application server built-in DataSource, Spring can access it via a Jndi proxy. Firstly configure a Jndi DataSource in appliation server GUI, then defines `JndiObjectFactoryBean` to access it via Jndi name.

```
@Bean
public DataSource prodDataSource() {
    JndiObjectFactoryBean ds = new JndiObjectFactoryBean();
    ds.setLookupOnStartup(true);
    ds.setJndiName("jdbc/postDS");
    ds.setCache(true);

    return (DataSource) ds.getObject();
}
```

The complete codes of `DataSouceConfig` .

```
@Configuration
public class DataSourceConfig {

    private static final String ENV_JDBC_PASSWORD = "jdbc.passwo
rd";
    private static final String ENV_JDBC_USERNAME = "jdbc.userna
```

```
me";
    private static final String ENV_JDBC_URL = "jdbc.url";

    @Inject
    private Environment env;

    @Bean
    @Profile("dev")
    public DataSource dataSource() {
        return new EmbeddedDatabaseBuilder()
                .setType(EmbeddedDatabaseType.H2)
                .build();
    }

    @Bean
    @Profile("staging")
    public DataSource testDataSource() {
        BasicDataSource bds = new BasicDataSource();
        bds.setDriverClassName("com.mysql.jdbc.Driver");
        bds.setUrl(env.getProperty(ENV_JDBC_URL));
        bds.setUsername(env.getProperty(ENV_JDBC_USERNAME));
        bds.setPassword(env.getProperty(ENV_JDBC_PASSWORD));
        return bds;
    }

    @Bean
    @Profile("prod")
    public DataSource prodDataSource() {
        JndiObjectFactoryBean ds = new JndiObjectFactoryBean();
        ds.setLookupOnStartup(true);
        ds.setJndiName("jdbc/postDS");
        ds.setCache(true);

        return (DataSource) ds.getObject();
    }

}
```

Three DataSouce beans are configured. Do not worry about the `@Profile` annotation, I will explain it in a *Spring Profile* related section for it.

# Configure JPA

JPA was proved a greate success in Java community, and it is wildly used in Java applications, including some desktop applications.

Spring embraces JPA specification in the first instance.

We have configured a DataSource, to support JPA in Spring, we need to configure a JPA specific `EntityManagerFactoryBean` and a `PlatformTransactionManager`.

Add `spring-orm` into *pom.xml*.

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
</dependency>
```

Use Hibernate as the JPA provider.

```
<!--java persistence API 2.1 -->
<dependency>
    <groupId>org.hibernate.javax.persistence</groupId>
    <artifactId>hibernate-jpa-2.1-api</artifactId>
    <version>1.0.0.Final</version>
</dependency>
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
</dependency>
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-entitymanager</artifactId>
    <exclusions>
        <exclusion>
            <groupId>cglib</groupId>
            <artifactId>cglib</artifactId>
        </exclusion>
        <exclusion>
            <groupId>dom4j</groupId>
            <artifactId>dom4j</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

Cooperate with Hibernate Core, we also use Bean Validation and `hibernate-validator` to generate database schema constraints.

```
<!--bean validation -->
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
</dependency>
<dependency>
    <groupId>javax.validation</groupId>
    <artifactId>validation-api</artifactId>
    <version>1.1.0.Final</version>
</dependency>
```

The following is the complete codes of `JpaConfig`.

```
@Configuration
@EnableTransactionManagement(mode = AdviceMode.ASPECTJ)
public class JpaConfig {

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

    private static final String ENV_HIBERNATE_DIALECT = "hiberna
te.dialect";
    private static final String ENV_HIBERNATE_HBM2DDL_AUTO = "hi
bernate.hbm2ddl.auto";
    private static final String ENV_HIBERNATE_SHOW_SQL = "hibern
ate.show_sql";
    private static final String ENV_HIBERNATE_FORMAT_SQL = "hibe
rnate.format_sql";

    @Inject
    private Environment env;

    @Inject
    private DataSource dataSource;

    @Bean
    public LocalContainerEntityManagerFactoryBean entityManagerF
actory() {
        LocalContainerEntityManagerFactoryBean emf = new LocalCo
ntainerEntityManagerFactoryBean();
        emf.setDataSource(dataSource);
        emf.setPackagesToScan("com.hantsylabs.restexample.spring
mvc");
        emf.setPersistenceProvider(new HibernatePersistenceProvi
der());
        emf.setJpaProperties(jpaProperties());
        return emf;
    }

    private Properties jpaProperties() {
        Properties extraProperties = new Properties();
```

```
        extraProperties.put(ENV_HIBERNATE_FORMAT_SQL, env.getPro
perty(ENV_HIBERNATE_FORMAT_SQL));
        extraProperties.put(ENV_HIBERNATE_SHOW_SQL, env.getPrope
rty(ENV_HIBERNATE_SHOW_SQL));
        extraProperties.put(ENV_HIBERNATE_HBM2DDL_AUTO, env.getP
roperty(ENV_HIBERNATE_HBM2DDL_AUTO));
        if (log.isDebugEnabled()) {
            log.debug(" hibernate.dialect @" + env.getProperty(E
NV_HIBERNATE_DIALECT));
        }
        if (env.getProperty(ENV_HIBERNATE_DIALECT) != null) {
            extraProperties.put(ENV_HIBERNATE_DIALECT, env.getPr
operty(ENV_HIBERNATE_DIALECT));
        }
        return extraProperties;
    }

    @Bean
    public PlatformTransactionManager transactionManager() {
        return new JpaTransactionManager(entityManagerFactory().
getObject());
    }
}
```

Generally, in a Java EE web application, JPA is activated by *META-INF/persistence.xml* file in the application archive.

The following is a classic JPA *persistence.xml*.

```
<persistence version="2.1"
   xmlns="http://xmlns.jcp.org/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
        http://xmlns.jcp.org/xml/ns/persistence
        http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.
xsd">
   <persistence-unit name="primary" transaction-type="RESOUECE_L
OCAL">

      <class>...Post</class>
      <none-jta-data-source/>
      <properties>
         <!-- Properties for Hibernate -->
         <property name="hibernate.hbm2ddl.auto" value="create-d
rop" />
         <property name="hibernate.show_sql" value="false" />
      </properties>
   </persistence-unit>
</persistence>
```

The transaction-type is `RESOUECE_LOCAL`, another available option is `JTA`. Most of the time, Spring applications is running in a Servlet container which does not support `JTA` by default.

You have to specify entity classes will be loaded and datasource here. Spring provides an alternative, `LocalEntityManagerFactoryBean` to simplify the configuration, there is a `setPackagesToScan` method provided to specify which packages will be scanned, and another `setDataSource` method to setup Spring DataSource configuration and no need to use database connection defined in the *persistence.xml* file at all.

More simply, `LocalContainerEntityManagerFactoryBean` does not need a *persistence.xml* file any more, and it builds JPA environment from ground. In above codes, we use `LocalContainerEntityManagerFactoryBean` to shrink JPA configuration.

Next, configure Spring Data JPA, add `spring-data-jpa` into pom.xml.

```
<!-- Spring Data -->
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-commons</artifactId>
</dependency>

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

Enable Spring Data JPA, use a standalone configuration class.

```
@Configuration
@EnableJpaRepositories(basePackages = {"com.hantsylabs.restexamp
le.springmvc"})
public class DataJpaConfig {

}
```

Spring Data Commons provides a series of `Repostory` APIs. `basePackages` in `@EnableJpaRepositories` will tell Spring Data JPA to scan `Repository` classes in these packages.

Do not forget to add `JpaConfig` and `DataJpaConfig` configuration classes into `getRootConfigClasses` method of `AppInitializer` .

# Configure Spring Security

Spring Security provides a specific `WebApplicationInitializer` to initialize Spring Security facilities.

```
@Order(1)
public class SecurityInitializer extends AbstractSecurityWebAppl
icationInitializer {

}
```

Similiar with `AbstractAnnotationConfigDispatcherServletInitializer`, it is a `WebApplicationInitializer` implementation, and aleady configured Spring Security filter chain for you.

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter
{
    @Override
    public void configure(WebSecurity web) throws Exception {
        web
            .ignoring()
            .antMatchers("/**/*.html", //
                         "/css/**", //
                         "/js/**", //
                         "/i18n/**",//
                         "/libs/**",//
                         "/img/**", //
                         "/webjars/**",//
                         "/ico/**");
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception
{
            http
```

```
                .authorizeRequests()
                .antMatchers("/api/**")
                .authenticated()
            .and()
                .authorizeRequests()
                .anyRequest()
                .permitAll()
                .and()
                    .sessionManagement()
                    .sessionCreationPolicy(SessionCreationPolicy
.STATELESS)
                .and()
                    .httpBasic()
                .and()
                    .csrf()
                    .disable();
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth)
            throws Exception {
        auth.inMemoryAuthentication()
                    .passwordEncoder(passwordEncoder())
                    .withUser("admin").password("test123").autho
rities("ROLE_ADMIN")
                    .and()
                        .withUser("test").password("test123").au
thorities("ROLE_USER");
    }


    @Bean
    @Override
    public AuthenticationManager authenticationManagerBean() thr
ows Exception {
        return super.authenticationManagerBean();
    }

    @Bean
    @Override
```

```
    public UserDetailsService userDetailsServiceBean() throws Ex
ception {
        return super.userDetailsServiceBean();
    }
}
```

`AuthenticationManagerBuilder` is the simplest entry to configure the essential security requirements. *InMemory* authentication is frequently used for demonstration or test purpose. In a real world project, it is better to implement a `UserDetailsService` to load users from database.

# Maven Profiles and Spring Profiles

Spring provides `@Profile` annotation to declare beans for specific profiles. You can specify a **spring.profiles.active** environment varible to activiate which beans will be used in Spring applications.

In this sample application, I would like use Maven profile to specify the **spring.profiles.active** for different development stage.

- For development stage, I would like an embeded database, such as H2. It is easy to user, and start up with a clean environment for test, I also like enable logging debug and get more log info.
- For staging stage, I would like use similar environment with produciton to run the application with a CI server, such as Jenkins, Travis CI, Circle CI etc.
- For produciton, enable all cache and performance optimization, use application server container managed datasource, only enable essential logging tracking etc.

We could plan some other profiles for UAT, etc.

In my angularjs-springmvc-sample, I defined 3 Maven profile for different purposes as described above.

```
<profiles>
    <profile>
        <id>dev</id>
        <activation>
            <activeByDefault>true</activeByDefault>
        </activation>
        <properties>

            <log4j.level>DEBUG</log4j.level>

            <spring.profiles.active>dev</spring.profiles.active>
            <!-- hibernate -->
            <hibernate.hbm2ddl.auto>create</hibernate.hbm2ddl.au
```

```
to>
            <hibernate.show_sql>true</hibernate.show_sql>
            <hibernate.format_sql>true</hibernate.format_sql>

            <!-- mail config -->
        </properties>
        <build>
            <resources>
                <resource>
                    <directory>src/main/resources-dev</directory
>
                    <filtering>true</filtering>
                </resource>
            </resources>
        </build>
    </profile>
    <profile>
        <id>staging</id>
        <properties>
            <jdbc.url><![CDATA[jdbc:mysql://localhost:3306/app]]
>
            </jdbc.url>
            <jdbc.username>root</jdbc.username>
            <jdbc.password></jdbc.password>

            <log4j.level>INFO</log4j.level>

            <spring.profiles.active>staging</spring.profiles.act
ive>

            <hibernate.hbm2ddl.auto>update</hibernate.hbm2ddl.au
to>
            <hibernate.show_sql>false</hibernate.show_sql>
            <hibernate.format_sql>false</hibernate.format_sql>
            <hibernate.dialect>org.hibernate.dialect.MySQL5Diale
ct</hibernate.dialect>

        </properties>
        <dependencies>
            <dependency>
```

```
                <groupId>mysql</groupId>
                <artifactId>mysql-connector-java</artifactId>
            </dependency>
        </dependencies>
        <build>
            <resources>
                <resource>
                    <directory>src/main/resources-staging</direc
tory>
                    <filtering>true</filtering>
                </resource>
            </resources>
        </build>
    </profile>
    <profile>
        <id>prod</id>
        <properties>
            <log4j.level>INFO</log4j.level>

            <spring.profiles.active>prod</spring.profiles.active
>

            <hibernate.hbm2ddl.auto>none</hibernate.hbm2ddl.auto
>
            <hibernate.show_sql>false</hibernate.show_sql>
            <hibernate.format_sql>false</hibernate.format_sql>
            <hibernate.dialect>org.hibernate.dialect.MySQL5Diale
ct</hibernate.dialect>
        </properties>
        <build>
            <resources>
                <resource>
                    <directory>src/main/resources-prod</director
y>
                    <filtering>true</filtering>
                </resource>
            </resources>
        </build>
    </profile>
</profiles>
```

In every Maven profiles, there is a *spring.profiles.active* property, its value will be filled in *applicaiton.properties* file and replaces the placeholder **@spring.profiles.active@** in this file, the properties files can be categoried in different files for varied purposes, such as database connection, global application settings etc. They are placed in the profile specific resource folder(defined in *resources* element in every profiles section).

Append a **-P** parameter to switch which Maven profile will be applied. eg.

```
mvn clean package -Pprod
```

The above command will package the applicatin for **prod** profile, it also apply **spring.profiles.active** value( `prod` ) for this application when it is running.

For exmaple, for the DataSource configuration in `DataSourceConfig`, the `@Profile("prod")` annotated bean will be activated.

```
@Bean
@Profile("prod")
public DataSource prodDataSource() {
    JndiObjectFactoryBean ds = new JndiObjectFactoryBean();
    ds.setLookupOnStartup(true);
    ds.setJndiName("jdbc/postDS");
    ds.setCache(true);

    return (DataSource) ds.getObject();
}
```

It uses the application server container managed DataSource for better performance and also easy to be monitored by application server console. Tomcat, Glassfish, Weblogic all provide friendly UI for administration.

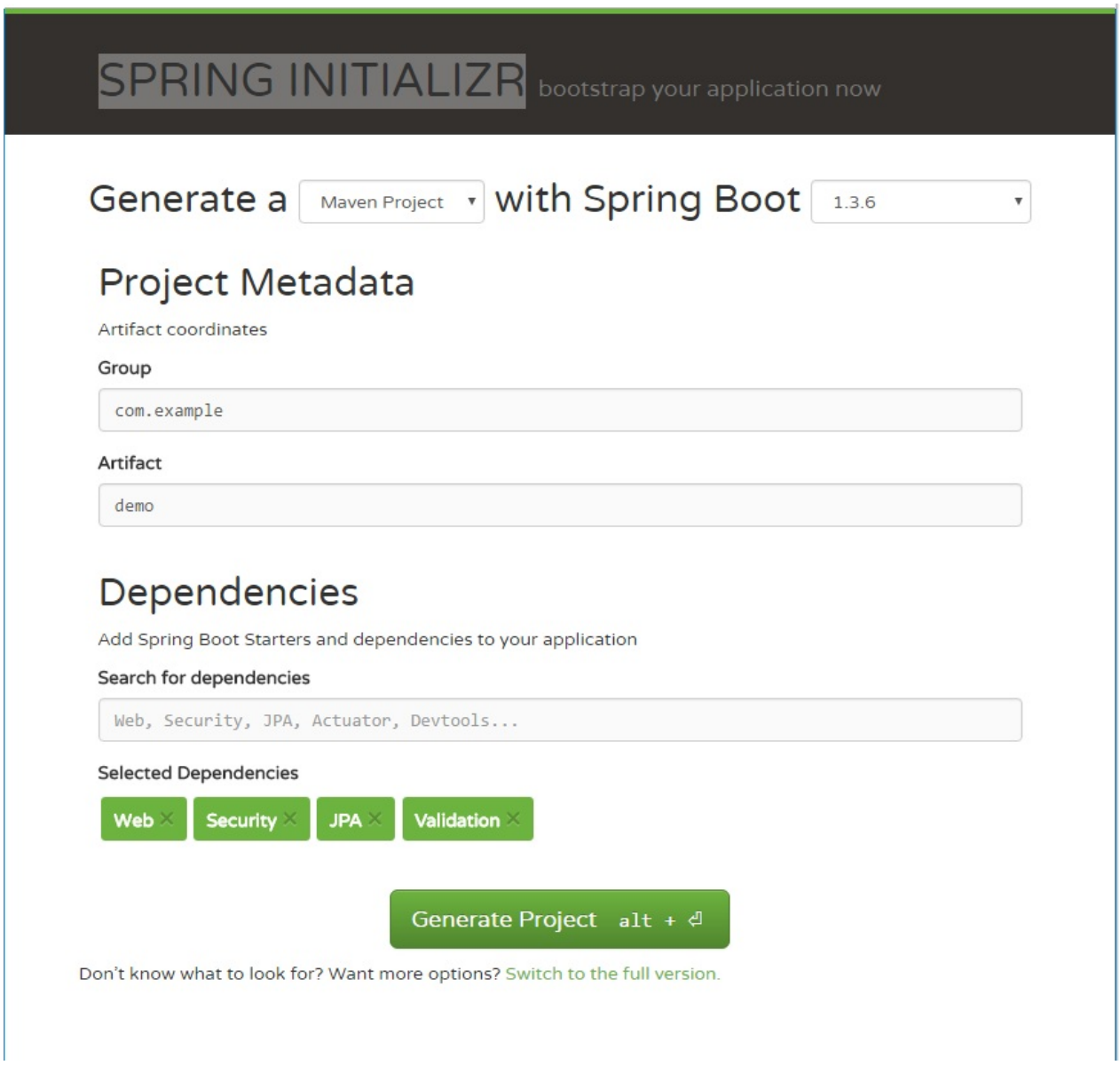
# Getting started with Spring Boot

Spring Boot provides simple autoconfigurations and losts of integrations for the thirdparty services. For developers, it decrease the configuration codes, and it is very quick to prototype an application. Spring Boot is the base of Spring microservice applications.

# Create project skeleton

SPRING INITIALIZR provides a visual web UI to start a Spring Boot project. You can select which starter will be added in the project you will create.

Open https://start.spring.io, search *Web*, *Secrity*, *JPA*, *Validation* in the **Dependencies** input box, select the items in the dropdown menus.




Then press `ALT+Enter` or click **Generate** button to download the generated codes in zip archive.

Extract the files into your local system.

As a start point, it only includes a few files.

- The Maven project configuration *pom.xml*, and several maven wrapper files which is like Gradle wrapper and use to download a specific maven for this project.
- A Spring Boot specific `Application` class as the application entry.
- A dummy test for the `Application` class.

Open the *pom.xml*, it looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="ht
tp://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>com.example</groupId>
    <artifactId>demo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>demo</name>
    <description>Demo project for Spring Boot</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.3.6.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourc
eEncoding>
        <java.version>1.8</java.version>
    </properties>
```

```
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId
>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-security</artifactId
>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-validation</artifact
Id>
        </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>

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

- The package type is **jar**, which means it will include an embedded tomcat at

build time. You can start the application via command line `java -jar app.jar`.
- The parent module is `spring-boot-starter-parent` which is a BOM(Bill of material) and includes the declaration of Spring Boot dependencies. Just add the dependencies you want to use under the `dependencies` node.
- Every starter will handle transitive dependencies. Besides those starters we selected, it also includes a starter for test purpose which will add the popular test dependencies transitively, such as *hamcrest*, *assertj*, *mockito* etc.
- `spring-boot-maven-plugin` allow you run the project in the embedded Tomcat.

Another important file is the entry class of this sample application.

```
@SpringBootApplication
public class DemoApplication {

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

`@SpringBootApplication` is a meta-annotation, which is a combined annotation with `@EnableAutoConfiguration`, `@ComponentScan` and `@SpringBootConfiguration`.

- `@EnableAutoConfiguration` enables the autoconfiguration detection by default.
- `@SpringBootConfiguration` is similiar with `@Configuration`, it indicates the application is a Spring Boot application, and only allow one `@SpringBootConfiguration` in an application.
- `@ComponentScan` defines the scope to find components, by default if not specify the package, it will scan the annotated class as base package. So it is recommended to put the `Application` in the root package.

Besides these, nothing! Where are the configuration files?

Spring Boot internally used plenty of auto-configuration mechanism to simplfy configurations for Spring developers. For this project, it configures a simple BASIC authentication by default. If you add a H2 database or other JDBC drivers, it will configure a datasource and transacation manager automaticially.

Till now, if you added some dependencies into `pom.xml`, you can start to code now. It is the quick way to prototype your application.

Although Spring Boot provides auto-configuration feature, but it does not prevent you to customize your configuration.

In the sample codes, there are some custom configuration classes.

# Configure DataSource

Instead of configuring `DataSource` in Java code.

Spring Boot provides built-in `application.properties` (or `applicatoin.yml` for YAML format) to configure `DataSource` in project specific file.

Create a *application.yml* in *src/main/resources* folder. It will override the default configuration.

```
server:
    port: 9000
    contextPath:

spring:
    profiles:
        active: dev
    devtools.restart.exclude: static/**,public/**
    datasource:
        dataSourceClassName: org.h2.jdbcx.JdbcDataSource
        url: jdbc:h2:mem:testdb;DB_CLOSE_DELAY=-1
        databaseName:
        serverName:
        username: sa
        password:

    jpa:
        database-platform: org.hibernate.dialect.H2Dialect
        database: H2
        openInView: false
        show_sql: true
        generate-ddl: true
        hibernate:
            ddl-auto:
            naming-strategy: org.hibernate.cfg.EJB3NamingStrateg
y
        properties:
            hibernate.cache.use_second_level_cache: true
```

```
            hibernate.cache.use_query_cache: false
            hibernate.generate_statistics: true
            hibernate.cache.region.factory_class: org.hibernate.
cache.internal.NoCachingRegionFactory

    data:
        jpa.repositories.enabled: true

    freemarker:
        check-template-location: false

    messages:
        basename: messages

logging:
    file: app.log
    level:
        root: INFO
        org.springframework.web: INFO
        com.hantsylabs.restexample.springmvc: DEBUG
```

This is a classic application configuration file in *YAML* format.

It is easy to understand.

**server.port** specifies the port number this application will serve at start up.

Under **spring** defines *DataSource*, *JPA*, *Spring Data JPA* etc.

**logging** configures logging level for packages.

**NOTE**: Spring Boot also supports *properties*, *groovy DSL* format for application configuration.

# Configure JPA

Make sure the following dependencies are added in your `dependency` section of the project `pom.xml` file.

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

It will add managed JPA, Spring Data JPA and Hibernate into your project.

We have already configured `DataSource`, and JPA and Spring Data JPA via `application.yml`.

You can also use Java code configuration to add some extra features.

```
@Configuration
@EnableTransactionManagement(mode = AdviceMode.ASPECTJ)
@EntityScan(basePackageClasses = {User.class, Jsr310JpaConverter
s.class})
@EnableJpaAuditing(auditorAwareRef = "auditor")
public class JpaConfig {

    @Bean
    public AuditorAware<User> auditor() {
        return () -> SecurityUtil.currentUser();
    }

}
```

In the above codes we add `@EnableTransactionManagement(mode = AdviceMode.ASPECTJ)` to enable transaction management and use `AspectJ` to weave transaction aspect into your business logic.

And `@EnableJpaAuditing(auditorAwareRef = "auditor")` enables the simple auditing features provided in Spring Data JPA. It requires a `AuditorAware` bean.

`@EntityScan(basePackageClasses = {User.class, Jsr310JpaConverters.class})` add the JPA entity scan scope, Java 8 DateTime support is added in Spring Data JPA via JPA 2.1 `AttributeConvertor` feature.

# Configure Spring Security

By default, Spring Boot will add BASIC authentication for your application. You can set the username and password in `application.yml` directly.

In a real world application, we would use DataSource driven configuration which you can use database to store user info.

Slightly changes the security configuration. Add a custom `WebSecurityConfigurerAdapter` bean is enough.

```
/**
 *
 * @author hantsy
 */
@Configuration
public class SecurityConfig {

    @Bean
    public WebSecurityConfigurerAdapter webSecurityConfigure(){
        return new WebSecurityConfigurerAdapter() {

            @Override
            protected void configure(HttpSecurity http) throws E
xception {
            // @formatter:off
                http
                    .authorizeRequests()
                    .antMatchers("/api/signup", "/api/users/user
name-check")
                    .permitAll()
                    .and()
                        .authorizeRequests()
                        .regexMatchers(HttpMethod.GET, "^/api/us
ers/[\\d]*(\\/)?$").authenticated()
                        .regexMatchers(HttpMethod.GET, "^/api/us
ers(\\/)?(\\?.+)?$").hasRole("ADMIN")
                        .regexMatchers(HttpMethod.DELETE, "^/api
```

```
/users/[\\d]*(\\/)?$").hasRole("ADMIN")
                        .regexMatchers(HttpMethod.POST, "^/api/u
sers(\\/)?$").hasRole("ADMIN")
                    .and()
                        .authorizeRequests()
                        .antMatchers("/api/**").authenticated()
                    .and()
                        .authorizeRequests()
                        .anyRequest().permitAll()
                    .and()
                        .sessionManagement()
                        .sessionCreationPolicy(SessionCreationPo
licy.STATELESS)
                    .and()
                        .httpBasic()
                    .and()
                        .csrf()
                        .disable();
            // @formatter:on
            }
        };
    }
}
```

To customize security, you could have to define your own `UserDetails` and `UserDetailsService` .

```
@Entity
@Table(name = "users")
public class User implements UserDetails, Serializable {

}
```

Create a JPA entity to emplement the `UserDetails` interface.

```
@Component
public class SimpleUserDetailsServiceImpl implements UserDetails
Service {

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

    private UserRepository userRepository;

    public SimpleUserDetailsServiceImpl(UserRepository userRepos
itory) {
        this.userRepository = userRepository;
    }

    @Override
    public UserDetails loadUserByUsername(String username) throws
 UsernameNotFoundException {
        User user = userRepository.findByUsername(username);
        if (user == null) {
            throw new UsernameNotFoundException("username not fo
und:" + username);
        }

        log.debug("found by username @" + username);

        return user;

    }

}
```

Define a `UserDetailsService`, which can be detected by the newest Spring Security, there is no need to wire the `UserDetailsService` with `AuthenticationManager` in configuration file. Check the Upgrade to Spring Boot 1.4 for more details.

# Configure Swagger

`SwaggerConfig` is no difference with before version.

Check out the source code to review it.

# Maven Profiles and Spring Profiles

Similar with the former villina version, we can define some Maven profiles to inject configuration for different project stages, such as *dev*, *staging*, *prod*. etc.

Open the project `pom.xml` file.

```
<profile>
    <id>dev</id>
    <activation>
        <activeByDefault>true</activeByDefault>
    </activation>
    <properties>
        <spring.profiles.active>dev</spring.profiles.active>
        <log4j.level>DEBUG</log4j.level>
    </properties>
    <build>
        <resources>
            <resource>
                <directory>src/main/resources-dev</directory>
                <filtering>true</filtering>
            </resource>
        </resources>
    </build>
</profile>
```

The above Maven profile(*dev*) will be activated by default, and it will add */src/main/reources-dev* as resource folder.

The *application.yml* under */src/main/reources-dev* will be packaged into the final package(jar or war).

We can define some other profile based application.yml for different stages. Every application.yml will add different configuration, such as in development stage, use a H2 database for easy testing, and in produciton profile, the application.yml will use a pool datasource for better performance at runtime.

The above approach combine Maven profiles and Spring profile to get clean and simple configuration for applications.

Alternatively, Spring Boot provides more powerful capability to switch profile via environment variables. You can package all profile based configuration in the same application package, and add a parameter to select a profile when the application is bootstraping.

```
java -jar ./app.jar -Dspring.profiles.active=staging
```

# Building REST API

As stated in before posts, we are going to build a Blog system.

To demonstrate REST API, we use a simple `Post` entity to persist blog entries, and expose the CRUD operations via REST APIs to client applications. As a REST API consumer, the client applications could be a website, a desktop application, or a mobile application.

Following the REST API convention and HTTP protocol specification, the post APIs can be designed as the following table.

| Uri | Http Method | Request | Response | Description |
|---|---|---|---|---|
| /posts | GET | | 200, [{'id':1, 'title'},{}] | Get all posts |
| /posts | POST | {'title':'test title','content':'test content'} | 201 | Create a new post |
| /posts/{id} | GET | | 200, {'id':1, 'title'} | Get a post by id |
| /posts/{id} | PUT | {'title':'test title','content':'test content'} | 204 | Update a post |
| /posts/{id} | DELETE | | 204 | Delete a post |

Next, we begin to create the domain models: `Post`.

## Modeling the blog application

As planned in Overview, there are some domain objects should be created for this blog sample application.

A `Post` model to store the blog entries posted by users. A `Comment` model to store the comments on a certain post. A `User` model to store users will user this blog application.

Every domain object should be identified. JPA entities satisfy this requirement. Every JPA entity has an `@Id` field as identifier.

A simple `Post` entity can be designated as the following. Besides id, it includes a `title` field , a `content` field, and `createdDate` timestamp, etc.

```
@Entity
@Table(name = "posts")
public class Post implements Serializable {

    @Id()
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Long id;

    @Column(name = "title")
    private String title;

    @Column(name = "content")
    @Size(max = 2000)
    private String content;

    @Column(name = "created_date")
    @Temporal(TemporalType.TIMESTAMP)
    private Date createdDate;

    //getters and setters, hashcode, equals, toString etc. are o
mitted
}
```

It is a standard JPA entity.

- An entity class must be annotated with `@Entity`
- An entity should implement `Serializable` interface
- An entity must include an identifier field(annotated with `@Id`), such as `id` of `Post`.
- An entity should have a default none-arguments constructor. By default, if there is no explict constructor declaration, there is an implict none-argument constructor. If there is a constructor accepts more than one arguments, you

have to add another none-argument explictly.

For example.

```
  public Post(String title, String content){}
  public Post(){}//must add this constructor explictly
```

Optionally, it is recommended to implement your own `equals` and `hashCode` methods for every entities, if there is a requirement to identify them in a collection.

For example, there are a post existed in a collection, adding another `Post` into the same collection should check the post existance firstly.

```
publc class PostCollection{
    private List<Post> posts=new ArrayList<>();

    public void addPost(Post p){
        if(!this.posts.contains(p)){
            this.posts.add(p);
        }
    }
}
```

The title field can be used to identify two posts in a collection, because they are not presisted in a persistent storage at the moment, `id` value are same--null.

Implements `equals` and `hashCode` with title field of `Post`.

```
public boolean equals(Object u){
    // omitted null check
    return this.title=(Post)u.getTitle();
}

public int hashCode(){
    return 17* title.hashCode();
}
```

When an entity instance is being persisted into a database table, the id will be filled.

In JPA specification, there is a sort of standard id generation strategies available.

By default, it is `AUTO`, which uses the database built-in id generation approache to assign an primary key to the inserted record.

**WARNING**: Every databases has its specific generation strategy, if you are building an application which will run across databases. `AUTO` is recommended.

Other id generation strategies include *TABLE*, *IDENTITY*. And JPA providers have their extensions, such as with Hibernate, you can use *uuid2* for PostgresSQL.

## Lombok

Lombok is a greate helper every Java developer should use in projects. Utilize Java annotation processor, it can generate getters, setters, equals, hashCode, toString and class constructor at compile runtime with some Lombok annotations.

Add `@Data` to `Post` class, you can remove all getters and setters, and `equals`, `hashcode`, `toString` methods. The code now looks more clean.

```
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
@Entity
@Table(name = "posts")
public class Post implements Serializable {
    @Id()
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Long id;

    @Column(name = "title")
    private String title;

    @Column(name = "content")
    @Size(max = 2000)
    private String content;

    @Column(name = "status")
    @Enumerated(value = EnumType.STRING)
    private Status status = Status.DRAFT;

    @Column(name = "created_date")
    @Temporal(TemporalType.TIMESTAMP)
    private Date createdDate;
}
```

There are some annotations provided in Lombok to archive this purpose.

`@Data` is a composite annotation which includes @Getter, @Setter, @EqualsAndHashCode,@ToString etc. `@Builder` will generate an inner Builder class in the hosted class which provides a fluent API to build an object.

Add lombok dependency in *pom.xml*.

```
<!--Lombok-->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>${lombok.version}</version>
</dependency>
```

If there are several JAP(Java annotaition processor) exist in the project, such as JPA metadata generator, it is better to add Lombok processor to maven compiler plugin.

For example.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.5.1</version>
    <configuration>
        <compilerArgument>-Xlint</compilerArgument>
        <annotationProcessors>
            <annotationProcessor>lombok.launch.AnnotationProcess
orHider$AnnotationProcessor</annotationProcessor>
            <annotationProcessor>org.hibernate.jpamodelgen.JPAMe
taModelEntityProcessor</annotationProcessor>
        </annotationProcessors>
    </configuration>
</plugin>
```

**NOTE**: If you are using Eclipse based IDE, such as Spring Tool Suite, or Intellij IDEA, you could have to install the Lombok plugin manually, check the Lombok download page for installation information. Luckily, NetBeans IDE can recognise the Lombok facilities automatcially.

Unlike JPA metadata generator which generates metedata source for JPA entities. Lombok modifies target classes directly.

Execute `javap Post.class` in command line, you can get the follwing info.

```
#>javap  classes\com\hantsylabs\restexample\springmvc\domain\Pos
t.class
Compiled from "Post.java"
public class com.hantsylabs.restexample.springmvc.domain.Post im
plements java.io.Serializable {
  public com.hantsylabs.restexample.springmvc.domain.Post(java.l
ang.String, java.lang.String);
  public static com.hantsylabs.restexample.springmvc.domain.Post
$PostBuilder builder();
  public java.lang.Long getId();
  public java.lang.String getTitle();
  public java.lang.String getContent();
  public com.hantsylabs.restexample.springmvc.domain.Post$Status
 getStatus();
  public com.hantsylabs.restexample.springmvc.domain.User getCre
atedBy();
  public java.time.LocalDateTime getCreatedDate();
  public com.hantsylabs.restexample.springmvc.domain.User getLas
tModifiedBy();
  public java.time.LocalDateTime getLastModifiedDate();
  public void setId(java.lang.Long);
  public void setTitle(java.lang.String);
  public void setContent(java.lang.String);
  public void setStatus(com.hantsylabs.restexample.springmvc.dom
ain.Post$Status);
  public void setCreatedBy(com.hantsylabs.restexample.springmvc.
domain.User);
  public void setCreatedDate(java.time.LocalDateTime);
  public void setLastModifiedBy(com.hantsylabs.restexample.sprin
gmvc.domain.User);
  public void setLastModifiedDate(java.time.LocalDateTime);
  public boolean equals(java.lang.Object);
  protected boolean canEqual(java.lang.Object);
  public int hashCode();
  public java.lang.String toString();
  public com.hantsylabs.restexample.springmvc.domain.Post();
  public com.hantsylabs.restexample.springmvc.domain.Post(java.l
ang.Long, java.lang.String, java.lang.String, com.hantsylabs.res
texample.springmvc.domain.Post$Status, com.hantsylabs.restexampl
```

```
e.springmvc.domain.User, java.time.LocalDateTime, com.hantsylabs
.restexample.springmvc.domain.User, java.time.LocalDateTime);
}
```

It prints all signatures of members of *Post.class*. As you see all essential methods(getters, setters, equals, hashCode, toString) have been added into the *Post.class*, and there is a *Post$Builder.class* file existed in the same folder, which is an inner class in the Post and implements the *Builder* pattern.

You can create a `Post` object using Post builder like this.

```
Post post = Post.builder()
            .title("title of my first post")
            .content("content of my first post")
            .build();
```

Compare to following legacy new an object, the Builder pattern is more friendly to developers, and codes become more readable.

```
Post post = new Post();
post.setTitle("title of my first post");
post.setContent("content of my first post");
```

## Model associations

Let's create other related models, `Comment` and `User`.

`Comment` class is associated with `Post` and `User`. Every comment should be belong to a post, and has an author( `User` ).

```
@Getter
@Setter
@ToString
@Builder
@NoArgsConstructor
@AllArgsConstructor
@Entity
```

```
@Table(name = "comments")
public class Comment implements Serializable {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    @Id()
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Long id;

    @Column(name = "content")
    private String content;

    @JoinColumn(name = "post_id")
    @ManyToOne()
    private Post post;

    @ManyToOne
    @JoinColumn(name = "created_by")
    @CreatedBy
    private User createdBy;

    @Column(name = "created_on")
    @CreatedDate
    private LocalDateTime createdDate;

    @Override
    public int hashCode() {
        int hash = 5;
        hash = 89 * hash + Objects.hashCode(this.content);
        return hash;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
```

```
        }
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        final Comment other = (Comment) obj;
        if (!Objects.equals(this.content, other.content)) {
            return false;
        }
        return true;
    }
}
```

`User` class contains fields of a user account, including username and password which used for authentication.

```
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
@Entity
@Table(name = "users")
public class User implements Serializable {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    @Id()
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Long id;

    @Column(name = "username")
    private String username;
```

```
    @Column(name = "password")
    private String password;

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

    @Column(name = "email")
    private String email;

    @Column(name = "role")
    private String role;

    @Column(name = "created_date")
    @CreatedDate
    private LocalDateTime createdDate;

    public String getName() {
        if (this.name == null || this.name.trim().length() == 0)
 {
            return this.username;
        }
        return name;
    }
}
```

A `Post` should have an author.

```
public class Post{

    @ManyToOne
    @JoinColumn(name = "created_by")
    @CreatedBy
    User createdBy;
}
```

# Data persistence with JPA

Generally, in Spring application, in order to make JPA work, you have to configure a `DataSource`, `EntityManagerFactory`, `TransactionManager`.

# JPA overview

JPA standardised Hibernate and it is part of Java EE specification since Java EE 5. Currently there are some popular JPA providers, such as Hibernate, OpenJPA, EclipseLink etc. EclipseLink is shipped wtih Glassfish, and Hibernate is included JBoss Wildfly/Redhat EAP.

In the above Modeling section, we have created models, which are JPA entities. In this section, let's see how to make it work.

## Configure DataSources

Like other ORM frameworks, you have to configure a DataSource.

The DataSourceConfig defines a series of DataSource for differnt profiles.

```
@Configuration
public class DataSourceConfig {

    private static final String ENV_JDBC_PASSWORD = "jdbc.passwo
rd";
    private static final String ENV_JDBC_USERNAME = "jdbc.userna
me";
    private static final String ENV_JDBC_URL = "jdbc.url";

    @Inject
    private Environment env;

    @Bean
    @Profile("dev")
    public DataSource dataSource() {
        return new EmbeddedDatabaseBuilder()
                .setType(EmbeddedDatabaseType.H2)
                .build();
    }
```

```
    @Bean
    @Profile("staging")
    public DataSource testDataSource() {
        BasicDataSource bds = new BasicDataSource();
        bds.setDriverClassName("com.mysql.jdbc.Driver");
        bds.setUrl(env.getProperty(ENV_JDBC_URL));
        bds.setUsername(env.getProperty(ENV_JDBC_USERNAME));
        bds.setPassword(env.getProperty(ENV_JDBC_PASSWORD));
        return bds;
    }

    @Bean
    @Profile("prod")
    public DataSource prodDataSource() {
        JndiObjectFactoryBean ds = new JndiObjectFactoryBean();
        ds.setLookupOnStartup(true);
        ds.setJndiName("jdbc/postDS");
        ds.setCache(true);

        return (DataSource) ds.getObject();
    }

}
```

In development stage("dev" profile is activated), using an embedded database is more easy to write tests, and speeds up development progress. In an integration server, it is recommended to run the appliation and integration tests on an environment close to production deployment. In a production environment,most of case, using a container managed datasource is effective.

The Spring profile can be activiated by an environment varible: `spring.profiles.active`. In our case, I used maven to set `spring.profiles.active` at compile time.

1. In the maven profile section, there is `spring.profiles.active` property defined. eg.

```
 <profile>
     <id>dev</id>
     <activation>
         <activeByDefault>true</activeByDefault>
     </activation>
     <properties>

         <log4j.level>DEBUG</log4j.level>

         <spring.profiles.active>dev</spring.profiles.active
>
         <!-- hibernate -->
         <hibernate.hbm2ddl.auto>create</hibernate.hbm2ddl.a
uto>
         <hibernate.show_sql>true</hibernate.show_sql>
         <hibernate.format_sql>true</hibernate.format_sql>

         <!-- mail config -->
     </properties>
     //...
 </profile>
```

2. Then used maven resource filter to replaced the placeholder defined in `app.properties` . Every maven profile could have a specific folder to hold the profiled based files. eg.

```
<profile>
    <id>dev</id>
    //...
    <build>
        <resources>
            <resource>
                <directory>src/main/resources-dev</directory
>
                <filtering>true</filtering>
            </resource>
        </resources>
    </build>
</profile>
```

3. After it is compiled, content of `app.properties` is filtered and replaced with the defined property.

```
 #app config properties
 spring.profiles.active=@spring.profiles.active@
```

Becomes:

```
 #app config properties
 spring.profiles.active=dev
```

4. In configuration class, add `PropertySource` to load the properties file.

```
 @PropertySource("classpath:/app.properties")
 @PropertySource(value = "classpath:/database.properties", i
gnoreResourceNotFound = true)
 public class AppConfig {

 }
```

NOTE: Read the Spring official document about Spring profile and Environment.

An example of JPA usage could like.

```
@Repository
@Transactional
public class PostRepository{

    @PersistenceContext
    private EntityManager em;
}
```

`@Repository` is an alias of `@Component`, `Transactional` is used to enable transaction on this bean. `@PersistenceContext` to inject an `EntityManager` to this bean. `EntityManager` provides a plent of methods to operate database.

For example,

`em.persist(Post)` to persist new entity. `em.merge(Post)` to merge the passed data into the entity existed and return a copy of the updated entity.

If you want to explore all methods provided in EntityManager, check EntityManager javadoc.

## Spring Data JPA

Spring Data JPA simplifies JPA, please read an early post I wrote to discuss this topic.

Use a `EnableJpaRepositories` to activiate Spring Data JPA. `basePackages` specifies packages will be scanned by Spring Data.

```
@Configuration
@EnableJpaRepositories(basePackages = {"com.hantsylabs.restexamp
le.springmvc"})
@EnableJpaAuditing(auditorAwareRef = "auditor")
public class JpaConfig {

    @Bean
    public AuditorAware<User> auditor() {
        return () -> SecurityUtil.currentUser();
    }

}
```

`EnableJpaAuditing` enable a simple auditing features provided in Spring Data JPA. There is some annotations are designated for it. Such as:

- `@CreatedBy`
- `@CreatedDate`
- `@LastModifiedBy`
- `@LastModifiedDate`

When `AuditingEntityListener` is activated globally in */META-INF/orm.xml*. Any fields annotated with above annotaitions will be filled automatically when the hosted entity is created and updated.

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings
    xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence/
orm http://xmlns.jcp.org/xml/ns/persistence/orm_2_1.xsd" version
="2.1">
    <persistence-unit-metadata>
        <persistence-unit-defaults>
            <entity-listeners>
                <entity-listener class="org.springframework.data
.jpa.domain.support.AuditingEntityListener" />
            </entity-listeners>
        </persistence-unit-defaults>
    </persistence-unit-metadata>
</entity-mappings>
```

`CreatedBy` and `LastModifiedBy` try to find a `AuditAware` bean and inject it into these fields at runtime. A simple implementation is getting the current principal from the Spring Security context.

## Repository

You can imagine a repository as a domain object collection, allow you retreive data from it or save change state back.

Spring Data Commons project defines a series of interfaces for common data operations for different storages, including NoSQL and RDBMS.

The top-level repository representation is the `Repository` interface.

`CrudRepository` subclasses from `Repository` and includes extra creating, retrieving, updating and deleting operations, aka CRUD.

A simple CRUD operations just need to create an interface extends `CrudRepository` .

```
interface MyPository extends CrudRepository{}
```

Create repository for Post.

```
public interface PostRepository extends JpaRepository<Post, Long
>,//
        JpaSpecificationExecutor<Post>{

}
```

`JpaRepository` is a JPA specific repository and derived from `PagingAndSortingRepository` which also subclasses from `CrudRepository` and provides pagination and sort capability. It accepts a `Pageable` object as method arguments, and can return a paged result with `Page` class.

`JpaSpecificationExecutor` is designated for JPA Criteria Query API, and provides type safe query instead of literal based JPQL.

For example, to search post by input keyword.

```
public static Specification<Post> filterByKeywordAndStatus(
        final String keyword,//
        final Post.Status status) {
    return (Root<Post> root, CriteriaQuery<?> query, CriteriaBui
lder cb) -> {
        List<Predicate> predicates = new ArrayList<>();
        if (StringUtils.hasText(keyword)) {
            predicates.add(
                cb.or(
                    cb.like(root.get(Post_.title), "%" + keyword
 + "%"),
                    cb.like(root.get(Post_.content), "%" + keywo
rd + "%")
                )
            );
        }

        if (status != null) {
            predicates.add(cb.equal(root.get(Post_.status), stat
us));
        }

        return cb.and(predicates.toArray(new Predicate[predicate
s.size()]));
    };
}
```

And you want to get a pageable result, you can use like this, just add a `Pageable` argument.

```
Page<Post> posts = postRepository.findAll(PostSpecifications.fil
terByKeywordAndStatus(q, status), page);
```

`page` argument is a `Pageable` object which can transfer pagination parameters from client request, and return result is a typed `Page` object, it includes the items of the current page and page naviation meta, such as total items, etc.

# Application Service

A service can delegate CRUD operations to repository, also act as gateway to other bound context, such as messageing, sending email, fire events etc.

```
@Service
@Transactional
public class BlogService {

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

    private PostRepository postRepository;

    @Inject
    public BlogService(PostRepository postRepository){
        this.postRepository = postRepository;
    }

    public Page<PostDetails> searchPostsByCriteria(String q, Pos
t.Status status, Pageable page) {

        log.debug("search posts by keyword@" + q + ", page @" +
page);

        Page<Post> posts = postRepository.findAll(PostSpecificat
ions.filterByKeywordAndStatus(q, status),
                page);

        log.debug("get posts size @" + posts.getTotalElements())
;

        return DTOUtils.mapPage(posts, PostDetails.class);
    }

    public PostDetails savePost(PostForm form) {

        log.debug("save post @" + form);
```

```
        Post post = DTOUtils.map(form, Post.class);

        Post saved = postRepository.save(post);

        log.debug("saved post is @" + saved);

        return DTOUtils.map(saved, PostDetails.class);

    }

    public PostDetails updatePost(Long id, PostForm form) {
        Assert.notNull(id, "post id can not be null");

        log.debug("update post @" + form);

        Post post = postRepository.findOne(id);
        DTOUtils.mapTo(form, post);

        Post saved = postRepository.save(post);

        log.debug("updated post@" + saved);

        return DTOUtils.map(saved, PostDetails.class);
    }

    public PostDetails findPostById(Long id) {

        Assert.notNull(id, "post id can not be null");

        log.debug("find post by id@" + id);

        Post post = postRepository.findOne(id);

        if (post == null) {
            throw new ResourceNotFoundException(id);
        }

        return DTOUtils.map(post, PostDetails.class);
    }
}
```

In this service there are some POJOs created for input request, response presentation etc.

`PostForm` gathers the user input data from client.

```
public class PostForm implements Serializable {

    @NotBlank
    private String title;

    private String content;
}
```

Validation annotations can be applied on it.

`PostDetails` represents the return result of a post details. Usually, it includes some info that should not include in the `PostForm`, such as id, timestamp etc.

```
public class PostDetails implements Serializable {

    private Long id;

    private String title;

    private String content;

    private String status;

    private Date createdDate;

}
```

Some exceptions are threw in the service if the input data can not satisfy the requirements. In the furthur post, I will focus on *exception handling* topic.

There is a `DTOUtils` which is responsible for data copy from one class to another class.

```
/**
 *
 * @author Hantsy Bai<hantsy@gmail.com>
 */
public final class DTOUtils {

    private static final ModelMapper INSTANCE = new ModelMapper(
);

    private DTOUtils() {
        throw new InstantiationError( "Must not instantiate this
 class" );
    }

    public static <S, T> T map(S source, Class<T> targetClass) {
        return INSTANCE.map(source, targetClass);
    }

    public static <S, T> void mapTo(S source, T dist) {
        INSTANCE.map(source, dist);
    }

    public static <S, T> List<T> mapList(List<S> source, Class<T
> targetClass) {
        List<T> list = new ArrayList<>();
        for (int i = 0; i < source.size(); i++) {
            T target = INSTANCE.map(source.get(i), targetClass);
            list.add(target);
        }

        return list;
    }

    public static <S, T> Page<T> mapPage(Page<S> source, Class<T
> targetClass) {
        List<S> sourceList = source.getContent();

        List<T> list = new ArrayList<>();
        for (int i = 0; i < sourceList.size(); i++) {
```

```
            T target = INSTANCE.map(sourceList.get(i), targetCla
ss);
            list.add(target);
        }

        return new PageImpl<>(list, new PageRequest(source.getNu
mber(), source.getSize(), source.getSort()),
                source.getTotalElements());
    }
}
```

It used the effort of ModelMapper project.

# Produces REST APIs with Spring MVC

Like other traditional action based framework, such as Apache Struts, etc, Spring MVC implements the standard MVC pattern. But against the benifit of IOC container, each parts of Spring MVC are not coupled, esp. view and view resolver are pluginable and can be configured.

For RESTful applications, JSON and XML are the commonly used exchange format, we do not need a template engine(such as Freemarker, Apache Velocity) for view, Spring MVC will detect HTTP headers, such as *Content Type*, *Accept Type*, etc. to determine how to produce corresponding view result. Most of the time, we do not need to configure the view/view resolver explictly. This is called *Content negotiation*. There is a `ContentNegotiationManager` bean which is responsible for *Content negotiation* and enabled by default in the latest version.

The configuration details are motioned in before posts. We are jumping to write `@Controller` to produce REST APIs.

Follows the REST design convention, create `PostController` to produce REST APIs.

```
@RestController
@RequestMapping(value = Constants.URI_API + Constants.URI_POSTS)
public class PostController {
```

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

    private BlogService blogService;

    @Inject
    public PostController(BlogService blogService) {
        this.blogService = blogService;
    }

    @RequestMapping(value = "", method = RequestMethod.GET)
    @ResponseBody
    public ResponseEntity<Page<PostDetails>> getAllPosts(
            @RequestParam(value = "q", required = false) String
keyword, //
            @RequestParam(value = "status", required = false) Po
st.Status status, //
            @PageableDefault(page = 0, size = 10, sort = "title"
, direction = Direction.DESC) Pageable page) {

        log.debug("get all posts of q@" + keyword + ", status @"
 + status + ", page@" + page);

        Page<PostDetails> posts = blogService.searchPostsByCrite
ria(keyword, status, page);

        log.debug("get posts size @" + posts.getTotalElements())
;

        return new ResponseEntity<>(posts, HttpStatus.OK);
    }

    @RequestMapping(value = "/{id}", method = RequestMethod.GET)
    @ResponseBody
    public ResponseEntity<PostDetails> getPost(@PathVariable("id
") Long id) {

        log.debug("get postsinfo by id @" + id);

        PostDetails post = blogService.findPostById(id);
```

```
        log.debug("get post @" + post);

        return new ResponseEntity<>(post, HttpStatus.OK);
    }

    @RequestMapping(value = "/{id}/comments", method = RequestMe
thod.GET)
    @ResponseBody
    public ResponseEntity<Page<CommentDetails>> getCommentsOfPos
t(
            @PathVariable("id") Long id,
            @PageableDefault(page = 0, size = 10, sort = "create
dDate", direction = Direction.DESC) Pageable page) {

        log.debug("get comments of post@" + id + ", page@" + pag
e);

        Page<CommentDetails> commentsOfPost = blogService.findCo
mmentsByPostId(id, page);

        log.debug("get post comment size @" + commentsOfPost.get
TotalElements());

        return new ResponseEntity<>(commentsOfPost, HttpStatus.O
K);
    }

    @RequestMapping(value = "", method = RequestMethod.POST)
    @ResponseBody
    public ResponseEntity<ResponseMessage> createPost(@RequestBo
dy @Valid PostForm post, BindingResult errResult) {

        log.debug("create a new post");
        if (errResult.hasErrors()) {
            throw new InvalidRequestException(errResult);
        }

        PostDetails saved = blogService.savePost(post);
```

```
        log.debug("saved post id is @" + saved.getId());

        HttpHeaders headers = new HttpHeaders();
        headers.setLocation(ServletUriComponentsBuilder.fromCurr
entContextPath()
                .path(Constants.URI_API + Constants.URI_POSTS +
"/{id}")
                .buildAndExpand(saved.getId())
                .toUri()
        );
        return new ResponseEntity<>(ResponseMessage.success("pos
t.created"), headers, HttpStatus.CREATED);
    }

    @RequestMapping(value = "/{id}", method = RequestMethod.DELE
TE)
    @ResponseBody
    public ResponseEntity<ResponseMessage> deletePostById(@PathV
ariable("id") Long id) {

        log.debug("delete post by id @" + id);

        blogService.deletePostById(id);

        return new ResponseEntity<>(ResponseMessage.success("pos
t.updated"), HttpStatus.NO_CONTENT);
    }

}
```

`@RestController` is a REST ready annotation, it is combined with `@Controller` and `@ResponseBody`.

`@RequestMapping` defines URL, HTTP methods etc are matched, the annotated method will handle the request.

*/api/posts* stands for a collection of `Post`, and */api/posts/{id}* is navigating to a specific Post which identified by id.

A *POST* method on */api/posts* is use for creating a new post, return HTTP status 201, and set HTTP header *Location* value to the new created post url if the creation is completed successfully.

*GET*, *PUT*, *DELETE* on */api/posts/{id}* performs retrieve, update, delete action on the certain post. *GET* will return a 200 HTTP status code, and *PUT*, *DELETE* return 204 if the operations are done as expected and there is no content body needs to be sent back to clients.

# Run

For none Spring Boot application, run it as a general web application in IDE.

Tomcat also provides maven plugin for those maven users.

```
<plugin>
    <groupId>org.apache.tomcat.maven</groupId>
    <artifactId>tomcat7-maven-plugin</artifactId>
    <version>2.2</version>
    <configuration>
        <path>/angularjs-springmvc-sample</path>
    </configuration>
</plugin>
```

Execute this in the command line to start this application.

```
mvn tomcat7:run
```

**NOTE**: The tomcat maven plugin development is not active, if you are using Servlet 3.1 features, you could have to use other plugin instead.

Jetty is the fastest embedded Servlet container and wildly used in development community.

```
<plugin>
    <groupId>org.eclipse.jetty</groupId>
    <artifactId>jetty-maven-plugin</artifactId>
    <version>9.3.7.v20160115</version>
    <configuration>
        <scanIntervalSeconds>10</scanIntervalSeconds>
        <stopPort>8005</stopPort>
        <stopKey>STOP</stopKey>
        <webApp>
            <contextPath>/angularjs-springmvc-sample</contextPat
h>
        </webApp>
    </configuration>
</plugin>
```

Execute the following command to run the application on an embedded Jetty server.

```
mvn jetty:run
```

Another frequently used is Cargo which provides support for all popular applcation servers, and ready for all hot build tools, such as Ant, Maven, Gradle etc.

```
<plugin>
    <groupId>org.codehaus.cargo</groupId>
    <artifactId>cargo-maven2-plugin</artifactId>
    <configuration>
        <container>
            <containerId>tomcat8x</containerId>
            <type>embedded</type>
        </container>

        <configuration>
            <properties>
                <cargo.servlet.port>9000</cargo.servlet.port>
                <cargo.logging>high</cargo.logging>
            </properties>
        </configuration>
    </configuration>
</plugin>
```

Execute the following command to run the application on an embedded Tomcat 8 server with the help of Cargo.

```
mvn verify cargo:run
```

For Spring boot application, it is simple, just run the application like this.

```
mvn spring-boot:run
```

By default, it uses Tomcat embedded server, but you can switch to Jetty and JBoss Undertow if you like. Check the Spring boot docs for details.

## Source Code

Check out sample codes from my github account.

```
git clone https://github.com/hantsy/angularjs-springmvc-sample
```

Or the Spring Boot version:

```
git clone https://github.com/hantsy/angularjs-springmvc-sample-b
oot
```

Read the live version of thess posts from Gitbook:Building RESTful APIs with Spring MVC.

# Handles Exceptions

In the real world applications, a user story can be described as different flows.

1. The execution works as expected and get success finally.
2. Some conditions are not satisfied and the flow should be intercepted and notify users.

For example, when a user tries to register an account in this application. The server side should check if the existance of the input username, if the username is taken by other users, the server should stop the registration flow and wraps the message into `UsernameWasTakenException` and throws it. Later the APIs should translate it to client friend message and reasonable HTTP status code, and finally they are sent to client and notify the user.

## Define Exceptions

Define an exeption which stands for the exception path. For example, `ResourceNotFoundException` indicates an resource is not found in the applicatin when query the resource by id.

```
public class ResourceNotFoundException extends RuntimeException
{

    private static final long serialVersionUID = 1L;

    private final Long id;

    public ResourceNotFoundException(Long id) {
        this.id = id;
    }

    public Long getId() {
        return id;
    }
}
```

## Throws exceptions in service

In our service, check the resource existance, and throws the `ResourceNotFoundException` when the resource is not found.

```
public PostDetails findPostById(Long id) {

    Assert.notNull(id, "post id can not be null");

    log.debug("find post by id@" + id);

    Post post = postRepository.findOne(id);

    if (post == null) {
        throw new ResourceNotFoundException(id);
    }

    return DTOUtils.map(post, PostDetails.class);
}
```

Here, we have defined ResourceNotFoundException as a `RuntimeException`, and there is no `throws` clause in the method declaration. Benefit from Spring IOC container, we do not need to use caught exception to force callers to handle it explictly. Alternatively, we can define a specific exception handler later to process it gracefully.

## Translates exceptions

Internally, Spring has a series of built-in `ExceptionTranslator`s to translate the exceptions to Spring declarative approaches, eg. all JDBC exceptions are translated to Spring defined exceptions(DataAccessException and its subclasses).

In the presentation layer, these exceptions can be caught and converted into user friendly message.

Spring provides a built-in `ResponseEntityExceptionHandler` to handle the exceptions and translate them into REST API friendly messages.

You can extend this class and override the default exeption hanlder methods, or add your exception hanlder to handle custom exceptions.

```
@ControllerAdvice(annotations = RestController.class)
public class RestExceptionHandler extends ResponseEntityExceptio
nHandler {

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

    @ExceptionHandler(value = {ResourceNotFoundException.class})
    @ResponseBody
    public ResponseEntity<ResponseMessage> handleResourceNotFoun
dException(ResourceNotFoundException ex, WebRequest request) {
        if (log.isDebugEnabled()) {
            log.debug("handling ResourceNotFoundException...");
        }
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
```

In the above code, `ResourceNotFoundException` is handled by the `RestExceptionHandler`, and send a 404 HTTP status code to the client.

## Handles bean validation failure

For user input form validation, most of time, we could needs the detailed info the validaiton cosntraints.

Spring supports JSR 303(Bean validation) natively, the bean validation constraints error can be gathered by `BindingResult` in the controller class.

```
public ResponseEntity<ResponseMessage> createPost(@RequestBody @
Valid PostForm post, BindingResult errResult) {

    log.debug("create a new post");
    if (errResult.hasErrors()) {
        throw new InvalidRequestException(errRusult);
    }
    //...
```

In the controller class, if the `BindingResult` has errors, then wraps the error info into an exception.

Handles it in the `RestExceptionHandler` .

```
@ExceptionHandler(value = {InvalidRequestException.class})
public ResponseEntity<ResponseMessage> handleInvalidRequestExcep
tion(InvalidRequestException ex, WebRequest req) {
    if (log.isDebugEnabled()) {
        log.debug("handling InvalidRequestException...");
    }

    ResponseMessage alert = new ResponseMessage(
        ResponseMessage.Type.danger,
        ApiErrors.INVALID_REQUEST,
        messageSource.getMessage(ApiErrors.INVALID_REQUEST, new
String[]{}, null));

    BindingResult result = ex.getErrors();

    List<FieldError> fieldErrors = result.getFieldErrors();

    if (!fieldErrors.isEmpty()) {
        fieldErrors.stream().forEach(e -> {
            alert.addError(e.getField(), e.getCode(), e.getDefau
ltMessage());
        });
    }

    return new ResponseEntity<>(alert, HttpStatus.UNPROCESSABLE_
ENTITY);
}
```

The detailed validation errors are wrapped as content and sent to the client, and a HTTP status to indicate the form data user entered is invalid.

## Exception and HTTP status

All Spring built-in exceptions have been handled and mapped to a HTTP status code. Read the `ResponseEntityExceptionHandler` code or javadoc for more details.

All business related exceptions should designed and converted to a valid HTTP status code and essential messages.

Some HTTP status codes are used frequently.

- 200 OK
- 201 Created
- 204 NO Content
- 400 Bad Resquest
- 401 Not Authoried
- 403 Forbidden
- 409 Conflict

More details about HTTP status code, please read https://httpstatuses.com/ or W3C HTTP Status definition.

# Source Code

Check out sample codes from my github account.

```
git clone https://github.com/hantsy/angularjs-springmvc-sample
```

Or the Spring Boot version:

```
git clone https://github.com/hantsy/angularjs-springmvc-sample-b
oot
```

Read the live version of thess posts from Gitbook:Building RESTful APIs with Spring MVC.

# Testing

Before release your applicaiton to the public world, you have to make sure it works as expected.

Testing is the most effective way to prove your codes correct.

## Test driven development

In the XP and Agile world, lots of developers are TDD advocate, and use it in daily work.

The basic flow of TDD can be summaried as:

1. Write a test first, then run test and get failure, failure info indicates what to do(You have not written any codes yet).
2. Code the implementation, and run test again and again, untill the test get passed.
3. Adjust the test to add more featurs, and refactor the codes, till all considerations are included.

But some developers prefer writing codes firstly and then write tests to verify them, it is OK. There is no policy to force you accept TDD. For a skilled developer, both are productive in work.

We have written some codes in the early posts, now it is time to add some test codes to show up how to test Spring components.

Spring provides a test context environment for developers, it supports JUnit and TestNG.

In this sample applcation, I will use JUnit as test runner, also use Mockito to test service in isolation, and use Rest Assured BDD like fluent APIs to test REST from client view.

## A simple POJO test

A classic JUnit test could look like this.

```
public class PostTest {

    public PostTest() {
    }

    @BeforeClass
    public static void setUpClass() {
    }

    @AfterClass
    public static void tearDownClass() {
    }

    private Post post;

    @Before
    public void setUp() {
        post = new Post();
        post.setTitle("test title");
        post.setContent("test content");
    }

    @After
    public void tearDown() {
        post = null;
    }

    /**
     * Test of getId method, of class Post.
     */
    @Test
    public void testPojo() {
        assertEquals("test title", post.getTitle());
        assertEquals("test content", post.getContent());
    }

}
```

`@BeforeClass` and `AfterClass` method must be *static*, these will be executed after the test class is constructed and before it is destoryed.

`@Before` and `@After` will be executed around a test case.

A test case is a method annotated with `@Test` .

Another annotation we always used is `@RunWith` , such as `@RunWith(SpringJUnit4ClassRunner.class)` , which will prepare specific test context for Spring tests.

Run the test in command line.

```
mvn test -Dtest=PostTest
```

You could see the following output summary for this test.

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.hantsylabs.restexample.springmvc.domain.PostTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
0.015 sec - in com.hantsylabs.restexample.springmvc.domain.PostT
est

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

`Post` is a simple POJO, does not depend on other dependencies.

# Test Service

`BlogService` depends on `PostRepository` , but most of time, we only want to check if the business logic and flow correct in the `BlogService` and assume the dependency `PostRepository` are always working as expected. Thus it is easy to focus on testing `BlogService` itself.

Mockito provides the simplest approaches to mock the dependencies, and setup the assumption, and provides an isolation environment to test `BlogService`.

## Isolate dependencies with Mockito

Mocks `PostRepository`, when invoke methods of `PostRepository`, return the dummy data we assumed. Mockito gives us a simple way to complete the assumption progress.

Create a `MockDataConfig` configuration class in test package.

```
@Configuration
public class MockDataConfig {

    @Bean
    public PostRepository postRepository() {
        final Post post = createPost();
        PostRepository posts = mock(PostRepository.class);
        when(posts.save(any(Post.class))).thenAnswer((Invocation
OnMock invocation) -> {
            Object[] args = invocation.getArguments();
            Post result = (Post) args[0];
            result.setId(post.getId());
            result.setTitle(post.getTitle());
            result.setContent(post.getContent());
            result.setCreatedDate(post.getCreatedDate());
            return result;
        });

        when(posts.findOne(1000L)).thenThrow(new ResourceNotFoun
dException(1000L));
        when(posts.findOne(1L)).thenReturn(post);
        when(posts.findAll(any(Specification.class), any(Pageabl
e.class))).thenReturn(new PageImpl(Arrays.asList(post), new Page
Request(0, 10), 1L));
        when(posts.findAll()).thenReturn(Arrays.asList(post));
        return posts;
    }
```

```
    @Bean
    public CommentRepository commentRepository() {
        return mock(CommentRepository.class);
    }

    @Bean
    public BlogService blogService(PostRepository posts, Comment
Repository comments){
        return new BlogService(posts, comments);
    }

    @Bean
    public Post createPost() {
        Post post = new Post();
        post.setCreatedDate(new Date());
        post.setId(1L);
        post.setTitle("First post");
        post.setContent("Content of my first post!");
        post.setCreatedDate(new Date());

        return post;
    }
}
```

We create mocked `PostRepository` in this configuration.

Create `MockBlogServiceTest` for `BlogService`, used `MockDataConfig` to load configurations.

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {MockDataConfig.class})
public class MockBlogServiceTest {

    private static final Logger LOG = LoggerFactory.getLogger(Mo
ckBlogServiceTest.class);

    @Inject
    private PostRepository postRepository;

    @Inject
    private CommentRepository commentRepository;

}
```

`PostRepository` and `CommentRepository` are mocked object defined in the `MockDataConfig`.

Add a test method in `MockBlogServiceTest`.

```
@Test
public void testSavePost() {
    PostForm form = new PostForm();
    form.setTitle("saving title");
    form.setContent("saving content");

    PostDetails details = blogService.savePost(form);

    LOG.debug("post details @" + details);
    assertNotNull("saved post id should not be null@", details.g
etId());
    assertTrue(details.getId() == 1L);

    //...
}
```

In the above test, asserts id value of the returned post is 1L.

Have a look at `MockDataConfig`, when calls save method of `PostRepository` and return a dummy post instance which id is 1L.

We can also test if the exception threw as expected in the `MockDataConfig`.

```
@Test(expected = ResourceNotFoundException.class)
public void testGetNoneExistingPost() {
    blogService.findPostById(1000L);
}
```

Run the test in command line tools.

```
mvn test -Dtest=MockBlogServiceTest
```

You will see the test result like.

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.hantsylabs.restexample.springmvc.test.MockBlogServic
eTest
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
1.943 sec - in com.hantsylabs.restexample.springmvc.test.MockBlo
gServiceTest

Results :

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
```

# Integration test

We have known `BlogService` works when we mocked the dependencies.

Now we write a test to check if it works against a real database.

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {AppConfig.class, DataSourceConf
```

```
ig.class, DataJpaConfig.class, JpaConfig.class})
public class BlogServiceTest {

    private static final Logger LOG = LoggerFactory.getLogger(Bl
ogServiceTest.class);

    @Inject
    private PostRepository postRepository;

    @Inject
    private BlogService blogService;

    private Post post;

    public BlogServiceTest() {
    }

    @Before
    public void setUp() {
        postRepository.deleteAll();
        post = postRepository.save(Fixtures.createPost("My first
 post", "content of my first post"));

        assertNotNull(post.getId());
    }

    @After
    public void tearDown() {
    }

    @Test
    public void testSavePost() {
        PostForm form = new PostForm();
        form.setTitle("saving title");
        form.setContent("saving content");

        PostDetails details = blogService.savePost(form);

        LOG.debug("post details @" + details);
        assertNotNull("saved post id should not be null@", detai
```

```
ls.getId());
        assertNotNull(details.getId());

        Page<PostDetails> allPosts = blogService.searchPostsByCr
iteria("", null, new PageRequest(0, 10));
        assertTrue(allPosts.getTotalElements() == 2);

        Page<PostDetails> posts = blogService.searchPostsByCrite
ria("first", Post.Status.DRAFT, new PageRequest(0, 10));
        assertTrue(posts.getTotalPages() == 1);
        assertTrue(!posts.getContent().isEmpty());
        assertTrue(Objects.equals(posts.getContent().get(0).getI
d(), post.getId()));

        PostForm updatingForm = new PostForm();
        updatingForm.setTitle("updating title");
        updatingForm.setContent("updating content");
        PostDetails updatedDetails = blogService.updatePost(post
.getId(), updatingForm);

        assertNotNull(updatedDetails.getId());
        assertTrue("updating title".equals(updatedDetails.getTit
le()));
        assertTrue("updating content".equals(updatedDetails.getC
ontent()));

    }

    @Test(expected = ResourceNotFoundException.class)
    public void testGetNoneExistingPost() {
        blogService.findPostById(1000L);
    }

}
```

In the `@Before` method, all Post data are cleared for each tests, and save a `Post` for further test assertion.

The above codes are similar with early Mockito version, the main difference is we have switched configuraitons to a real database. Check the `@ContextConfiguration` annotated on `BlogServiceTest`.

Run the test.

```
mvn test -Dtest=BlogServiceTest
```

The test result should be shown as below.

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.hantsylabs.restexample.springmvc.test.BlogServiceTes
t
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
5.908 sec - in com.hantsylabs.restexample.springmvc.test.BlogSer
viceTest

Results :

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
```

# Test Controller

Spring provides a sort of mock APIs to emulate a Servlet container environment, thus it is possbile to test MVC related feature without a real container.

## Use MockMvc with mocked service

MockMvc does not need a Servlet container, but can test most of the Controller features.

Like the former `MockBlogServiceTest`, we can mock the controller's dependencies, thus is no need to load Spring configuraitons.

```
@RunWith(MockitoJUnitRunner.class)
```

```
public class MockPostControllerTest {

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

    private MockMvc mvc;

    ObjectMapper objectMapper = new ObjectMapper();

    @Mock
    private BlogService blogService;

    @Mock
    Pageable pageable = mock(PageRequest.class);

    @InjectMocks
    PostController postController;

    @BeforeClass
    public static void beforeClass() {
        log.debug("==================before class==============
==========");
    }

    @AfterClass
    public static void afterClass() {
        log.debug("==================after class===============
=========");
    }

    @Before
    public void setup() {
        log.debug("==================before test case==========
==============");
        Mockito.reset();
        MockitoAnnotations.initMocks(this);
        mvc = standaloneSetup(postController)
                .setCustomArgumentResolvers(new PageableHandlerM
ethodArgumentResolver())
                .setViewResolvers(new ViewResolver() {
```

```
                    @Override
                    public View resolveViewName(String viewName,
 Locale locale) throws Exception {
                        return new MappingJackson2JsonView();
                    }
                })
                .build();
    }

    @After
    public void tearDown() {
        log.debug("==================after test case============
=============");
    }

    @Test
    public void savePost() throws Exception {
        PostForm post = Fixtures.createPostForm("First Post", "C
ontent of my first post!");

        when(blogService.savePost(any(PostForm.class))).thenAnsw
er(new Answer<PostDetails>() {
            @Override
            public PostDetails answer(InvocationOnMock invocatio
n) throws Throwable {
                PostForm fm = (PostForm) invocation.getArgumentA
t(0, PostForm.class);

                PostDetails result = new PostDetails();
                result.setId(1L);
                result.setTitle(fm.getTitle());
                result.setContent(fm.getContent());
                result.setCreatedDate(new Date());

                return result;
            }
        });
        mvc.perform(post("/api/posts").contentType(MediaType.APP
LICATION_JSON).content(objectMapper.writeValueAsString(post)))
                .andExpect(status().isCreated());
```

```
        verify(blogService, times(1)).savePost(any(PostForm.clas
s));
        verifyNoMoreInteractions(blogService);
    }

    @Test
    public void retrievePosts() throws Exception {
        PostDetails post1 = new PostDetails();
        post1.setId(1L);
        post1.setTitle("First post");
        post1.setContent("Content of first post");
        post1.setCreatedDate(new Date());

        PostDetails post2 = new PostDetails();
        post2.setId(2L);
        post2.setTitle("Second post");
        post2.setContent("Content of second post");
        post2.setCreatedDate(new Date());

        when(blogService.searchPostsByCriteria(anyString(), any(
Post.Status.class), any(Pageable.class)))
                .thenReturn(new PageImpl(Arrays.asList(post1, po
st2), new PageRequest(0, 10, Direction.DESC, "createdDate"), 2))
;

        MvcResult response = mvc.perform(get("/api/posts?q=test&
page=0&size=10"))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.content[*].id", hasItem(1
)))
                .andExpect(jsonPath("$.content[*].title", hasIte
m("First post")))
                .andReturn();

        verify(blogService, times(1))
                .searchPostsByCriteria(anyString(), any(Post.Sta
tus.class), any(Pageable.class));
        verifyNoMoreInteractions(blogService);
        log.debug("get posts result @" + response.getResponse().
```

```
getContentAsString());
    }

    @Test
    public void retrieveSinglePost() throws Exception {

        PostDetails post1 = new PostDetails();
        post1.setId(1L);
        post1.setTitle("First post");
        post1.setContent("Content of first post");
        post1.setCreatedDate(new Date());

        when(blogService.findPostById(1L)).thenReturn(post1);

        mvc.perform(get("/api/posts/1").accept(MediaType.APPLICA
TION_JSON))
                .andExpect(status().isOk())
                .andExpect(content().contentType("application/js
on;charset=UTF-8"))
                .andExpect(jsonPath("id").isNumber());

        verify(blogService, times(1)).findPostById(1L);
        verifyNoMoreInteractions(blogService);
    }

    @Test
    public void removePost() throws Exception {
        when(blogService.deletePostById(1L)).thenReturn(true);
        mvc.perform(delete("/api/posts/{id}", 1L))
                .andExpect(status().isNoContent());

        verify(blogService, times(1)).deletePostById(1L);
        verifyNoMoreInteractions(blogService);
    }

    @Test()
    public void notFound() {
        when(blogService.findPostById(1000L)).thenThrow(new Reso
urceNotFoundException(1000L));
        try {
```

```
            mvc.perform(get("/api/posts/1000").accept(MediaType.
APPLICATION_JSON))
                    .andExpect(status().isNotFound());
        } catch (Exception ex) {
            log.debug("exception caught @" + ex);
        }
    }

}
```

In the `setup` method, `mvc = standaloneSetup(postController)` is trying to setup a `MockMvc` for controller. The test codes are easy to understand.

## MockMvc with a real database

We changed a little on the above tests, replace the mocked service with the real configurations. Thus the tests will run against a real database, but still in mock mvc environment.

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {AppConfig.class, Jackson2Object
MapperConfig.class, DataSourceConfig.class, JpaConfig.class, Dat
aJpaConfig.class, WebConfig.class})
@WebAppConfiguration
public class PostControllerTest {

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

    @Inject
    WebApplicationContext wac;

    @Inject
    ObjectMapper objectMapper;

    @Inject
    private PostRepository postRepository;
```

```
    private MockMvc mvc;

    private Post post;

    @BeforeClass
    public static void beforeClass() {
        log.debug("==================before class==============
==========");
    }

    @AfterClass
    public static void afterClass() {
        log.debug("==================after class===============
=========");
    }

    @Before
    public void setup() {
        log.debug("==================before test case==========
==============");
        mvc = webAppContextSetup(this.wac).build();

        postRepository.deleteAll();
        post = postRepository.save(Fixtures.createPost("My first
 post", "content of my first post"));
    }

    @After
    public void tearDown() {
        log.debug("==================after test case===========
=============");
    }

    @Test
    public void savePost() throws Exception {
        PostForm post = Fixtures.createPostForm("First Post", "C
ontent of my first post!");

        mvc.perform(post("/api/posts").contentType(MediaType.APP
LICATION_JSON).content(objectMapper.writeValueAsString(post)))
```

```
                .andExpect(status().isCreated());

    }

    @Test
    public void retrievePosts() throws Exception {

        MvcResult response = mvc.perform(get("/api/posts?q=first
&page=0&size=10"))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.content[0].id", is(post.g
etId().intValue())))
                .andExpect(jsonPath("$.content[0].title", is("My
 first post")))
                .andReturn();

        log.debug("get posts result @" + response.getResponse().
getContentAsString());
    }

    @Test
    public void retrieveSinglePost() throws Exception {

        mvc.perform(get("/api/posts/{id}", post.getId()).accept(
MediaType.APPLICATION_JSON))
                .andExpect(status().isOk())
                .andExpect(content().contentType("application/js
on"))
                .andExpect(jsonPath("$.id").isNumber())
                .andExpect(jsonPath("$.title", is("My first post
")));

    }

    @Test
    public void removePost() throws Exception {
        mvc.perform(delete("/api/posts/{id}", post.getId()))
                .andExpect(status().isNoContent());
    }
```

```
    @Test()
    public void notFound() {
        try {
            mvc.perform(get("/api/posts/1000").accept(MediaType.
APPLICATION_JSON))
                    .andExpect(status().isNotFound());
        } catch (Exception ex) {
            log.debug("exception caught @" + ex);
        }
    }

}
```

In this test class, the Mockito codes are replaced with Spring test, and load the configuraitons defined in this project. It is close to the final production environment, except there is not a real Servlet container.

## Test REST API as the client view

OK, now try to verify everything works in a real container.

```
@RunWith(BlockJUnit4ClassRunner.class)
public class IntegrationTest {

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

    private static final String BASE_URL = "http://localhost:808
0/angularjs-springmvc-sample/";

    private RestTemplate template;

    @BeforeClass
    public static void init() {
        log.debug("==================before class==============
==========");
    }

    @Before
```

```
    public void beforeTestCase() {
        log.debug("==================before test case==========
==============");
        template = new BasicAuthRestTemplate("admin", "test123")
;
    }

    @After
    public void afterTestCase() {
        log.debug("==================after test case===========
=============");
    }

    @Test
    public void testPostCrudOperations() throws Exception {
        PostForm newPost = Fixtures.createPostForm("My first pos
t", "content of my first post");
        String postsUrl = BASE_URL + "api/posts";

        ResponseEntity<Void> postResult = template.postForEntity
(postsUrl, newPost, Void.class);
        assertTrue(HttpStatus.CREATED.equals(postResult.getStatu
sCode()));
        String createdPostUrl = postResult.getHeaders().getLocat
ion().toString();
        assertNotNull("created post url should be set", createdP
ostUrl);

        ResponseEntity<Post> getPostResult = template.getForEnti
ty(createdPostUrl, Post.class);
        assertTrue(HttpStatus.OK.equals(getPostResult.getStatusC
ode()));
        log.debug("post @" + getPostResult.getBody());
        assertTrue(getPostResult.getBody().getTitle().equals(new
Post.getTitle()));

        ResponseEntity<Void> deleteResult = template.exchange(cr
eatedPostUrl, HttpMethod.DELETE, null, Void.class);
        assertTrue(HttpStatus.NO_CONTENT.equals(deleteResult.get
StatusCode()));
```

```
    }

    @Test
    public void noneExistingPost() throws Exception {
        String noneExistingPostUrl = BASE_URL + "api/posts/1000"
;
        try {
            template.getForEntity(noneExistingPostUrl, Post.clas
s);
        } catch (HttpClientErrorException e) {
            assertTrue(HttpStatus.NOT_FOUND.equals(e.getStatusCo
de()));
        }
    }
}
```

`RestTemplate` is use for interaction with remote REST API, this test acts as a remote client, and shake hands with our backend through REST APIs.

`BasicAuthRestTemplate` is a helper class to process *BASIC* authentication.

```
public class BasicAuthRestTemplate extends RestTemplate {

    public BasicAuthRestTemplate(String username, String passwor
d) {
        addAuthentication(username, password);
    }

    private void addAuthentication(String username, String passw
ord) {
        if (username == null) {
            return;
        }
        List<ClientHttpRequestInterceptor> interceptors = Collec
tions
                .<ClientHttpRequestInterceptor>singletonList(
                        new BasicAuthorizationInterceptor(userna
me, password));
        setRequestFactory(new InterceptingClientHttpRequestFacto
```

```
ry(getRequestFactory(),
                interceptors));
    }

    private static class BasicAuthorizationInterceptor implement
s
            ClientHttpRequestInterceptor {

        private final String username;

        private final String password;

        public BasicAuthorizationInterceptor(String username, St
ring password) {
            this.username = username;
            this.password = (password == null ? "" : password);
        }

        @Override
        public ClientHttpResponse intercept(HttpRequest request,
 byte[] body,
                ClientHttpRequestExecution execution) throws IOE
xception {
            byte[] token = Base64.getEncoder().encode(
                    (this.username + ":" + this.password).getByt
es());
            request.getHeaders().add("Authorization", "Basic " +
 new String(token));
            return execution.execute(request, body);
        }

    }
}
```

To run this test successfully, you have to configure *maven-failsafe-plugin* to set up a servlet container.

- Start up container before test is running
- Shutdown the servlet container after the test is completed

org.apache.maven.plugins maven-surefire-plugin 2.19 true true **/*IntegrationTest*

Excludes the `IntegrationTest` in the *maven-surefire-plugin*.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-failsafe-plugin</artifactId>
    <version>2.12.4</version>
    <configuration>
        <includes>
            <include>**/*IntegrationTest*</include>
        </includes>
    </configuration>
    <executions>
        <execution>
            <id>integration-test</id>
            <goals>
                <goal>integration-test</goal>
            </goals>
        </execution>
        <execution>
            <id>verify</id>
            <goals>
                <goal>verify</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Filter the `IntegrationTest` in the *maven-failsafe-plugin*. Here I configured jetty as servlet container to run the test.

```
<plugin>
    <groupId>org.eclipse.jetty</groupId>
    <artifactId>jetty-maven-plugin</artifactId>
    <version>9.3.7.v20160115</version>
    <configuration>
        <scanIntervalSeconds>10</scanIntervalSeconds>
        <stopPort>8005</stopPort>
        <stopKey>STOP</stopKey>
        <webApp>
            <contextPath>/angularjs-springmvc-sample</contextPat
h>
        </webApp>
    </configuration>
    <executions>
        <execution>
            <id>start-jetty</id>
            <phase>pre-integration-test</phase>
            <goals>
                <goal>stop</goal>
                <goal>start</goal>
            </goals>
            <configuration>
                <scanIntervalSeconds>0</scanIntervalSeconds>
                <daemon>true</daemon>
            </configuration>
        </execution>
        <execution>
            <id>stop-jetty</id>
            <phase>post-integration-test</phase>
            <goals>
                <goal>stop</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

In the *pre-integration-test* phase, check if the jetty is running and starts up it, in `post-integration-test` phase, shutdown the container.

Run the `IntegrationTest` in command line.

```
mvn clean verify
```

In the console, after all unit tess are done, it will start jetty and deploy the project war into jetty and run the `IntegrationTest` on it.

```
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
2.847 sec

Results :

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0

[WARNING] File encoding has not been set, using platform encodin
g Cp1252, i.e. build is platform dependent!
[INFO]
[INFO] --- jetty-maven-plugin:9.3.7.v20160115:stop (stop-jetty)
@ angularjs-springmvc-sample ---
[INFO]
[INFO] --- maven-failsafe-plugin:2.12.4:verify (verify) @ angula
rjs-springmvc-sample ---
```

As you see in the console, after the test is done, it is trying to shutdown jetty.

## Rest Assured

Rest Assured provides BDD like syntax, such as *given*, *when*, *then*, it is friendly for those familiar with BDD.

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
@Slf4j
public class RestAssuredApplicationTest extends WebIntegrationTe
stBase {
```

```
    @Before
    public void beforeTest() {
        super.setup();
        RestAssured.port = port;
    }

    @Test
    public void testDeletePostNotExisted() {
        String location = "/api/posts/1000";

        given()
                .auth().basic(USER_NAME, PASSWORD)
                .contentType(ContentType.JSON)
                .when()
                .delete(location)
                .then()
                .assertThat()
                .statusCode(HttpStatus.SC_NOT_FOUND);
    }

    @Test
    public void testGetPostNotExisted() {
        String location = "/api/posts/1000";

        given()
                .auth().basic(USER_NAME, PASSWORD)
                .contentType(ContentType.JSON)
                .when()
                .get(location)
                .then()
                .assertThat()
                .statusCode(HttpStatus.SC_NOT_FOUND);
    }

    @Test
    public void testPostFormInValid() {
        PostForm form = new PostForm();

        given()
                .auth().basic(USER_NAME, PASSWORD)
```

```
                .body(form)
                .contentType(ContentType.JSON)
                .when()
                .post("/api/posts")
                .then()
                .assertThat()
                .statusCode(HttpStatus.SC_BAD_REQUEST);
    }

    @Test
    public void testPostCRUD() {
        PostForm form = new PostForm();
        form.setTitle("test title");
        form.setContent("test content");

        Response response = given()
                .auth().basic(USER_NAME, PASSWORD)
                .body(form)
                .contentType(ContentType.JSON)
                .when()
                .post("/api/posts")
                .then()
                .assertThat()
                .statusCode(HttpStatus.SC_CREATED)
                .and()
                .header("Location", containsString("/api/posts/"
))
                .extract().response();

        String location = response.header("Location");

        log.debug("header location value @" + location);

        given().auth().basic(USER_NAME, PASSWORD)
                .contentType(ContentType.JSON)
                .when()
                .get(location)
                .then()
                .assertThat()
                .body("title", is("test title"))
```

```
                .body("content", is("test content"));

        PostForm updateForm = new PostForm();
        updateForm.setTitle("test udpate title");
        updateForm.setContent("test update content");

        given()
                .auth().basic(USER_NAME, PASSWORD)
                .body(updateForm)
                .contentType(ContentType.JSON)
                .when()
                .put(location)
                .then()
                .assertThat()
                .statusCode(HttpStatus.SC_NO_CONTENT);

        given().auth().basic(USER_NAME, PASSWORD)
                .contentType(ContentType.JSON)
                .when()
                .get(location)
                .then()
                .assertThat()
                .body("title", is("test udpate title"))
                .body("content", is("test update content"));

        given()
                .auth().basic(USER_NAME, PASSWORD)
                .contentType(ContentType.JSON)
                .when()
                .delete(location)
                .then()
                .assertThat()
                .statusCode(HttpStatus.SC_NO_CONTENT);

        given().auth().basic(USER_NAME, PASSWORD)
                .contentType(ContentType.JSON)
                .when()
                .get(location)
                .then()
                .assertThat()
```

```
                .statusCode(HttpStatus.SC_NOT_FOUND);

    }

}
```

This test is also run as client, and interacts with backend via REST API.

The above Rest Assured sample codes are available in the Spring Boot version, check out the codes and experience yourself.

It also includes a simple JBehave sample, if you are a JBehave user, you maybe interested in it.

# Source Code

Check out sample codes from my github account.

```
git clone https://github.com/hantsy/angularjs-springmvc-sample
```

Or the Spring Boot version:

```
git clone https://github.com/hantsy/angularjs-springmvc-sample-b
oot
```

Read the live version of thess posts from Gitbook:Building RESTful APIs with Spring MVC.

# Visualizes REST APIs with Swagger

Swagger is widely used for visualizing APIs, and with Swagger UI it provides online sandbox for frontend developers.

## Visualizes REST APIs

SpringFox project provides Swagger support for Spring based REST APIs.

1. Add springfox to dependencies.

```
<!-- SpringFox Swagger UI -->
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>${springfox.version}</version>
</dependency>

<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>${springfox.version}</version>
</dependency>
```

   `springfox-swagger-ui` provides static Javascript UI for visualizing the Swagger schema definitions.

2. Add a `@Configuration` class to enable Swagger.

```
@Configuration
@EnableSwagger2
public class SwaggerConfig {

    @Bean
    public Docket postsApi() {
        return new Docket(DocumentationType.SWAGGER_2)
                .groupName("public-api")
                .apiInfo(apiInfo())
                .select()
                .paths(postPaths())
                .build();
    }

    private Predicate<String> postPaths() {
        return or(
                regex("/api/posts.*"),
                regex("/api/comments.*")
        );
    }

    private ApiInfo apiInfo() {
        return new ApiInfoBuilder()
                .title("SpringMVC Example API")
                .description("SpringMVC Example API referen
ce for developers")
                .termsOfServiceUrl("http://hantsy.blogspot.
com")
                .contact("Hantsy Bai")
                .license("Apache License Version 2.0")
                .licenseUrl("https://github.com/springfox/s
pringfox/blob/master/LICENSE")
                .version("2.0")
                .build();
    }

}
```

When the application starts up, it will scan all Controllers and generate Swagger schema definition at runtime, Swagger UI will read definitions and render user friendly UI for REST APIs.

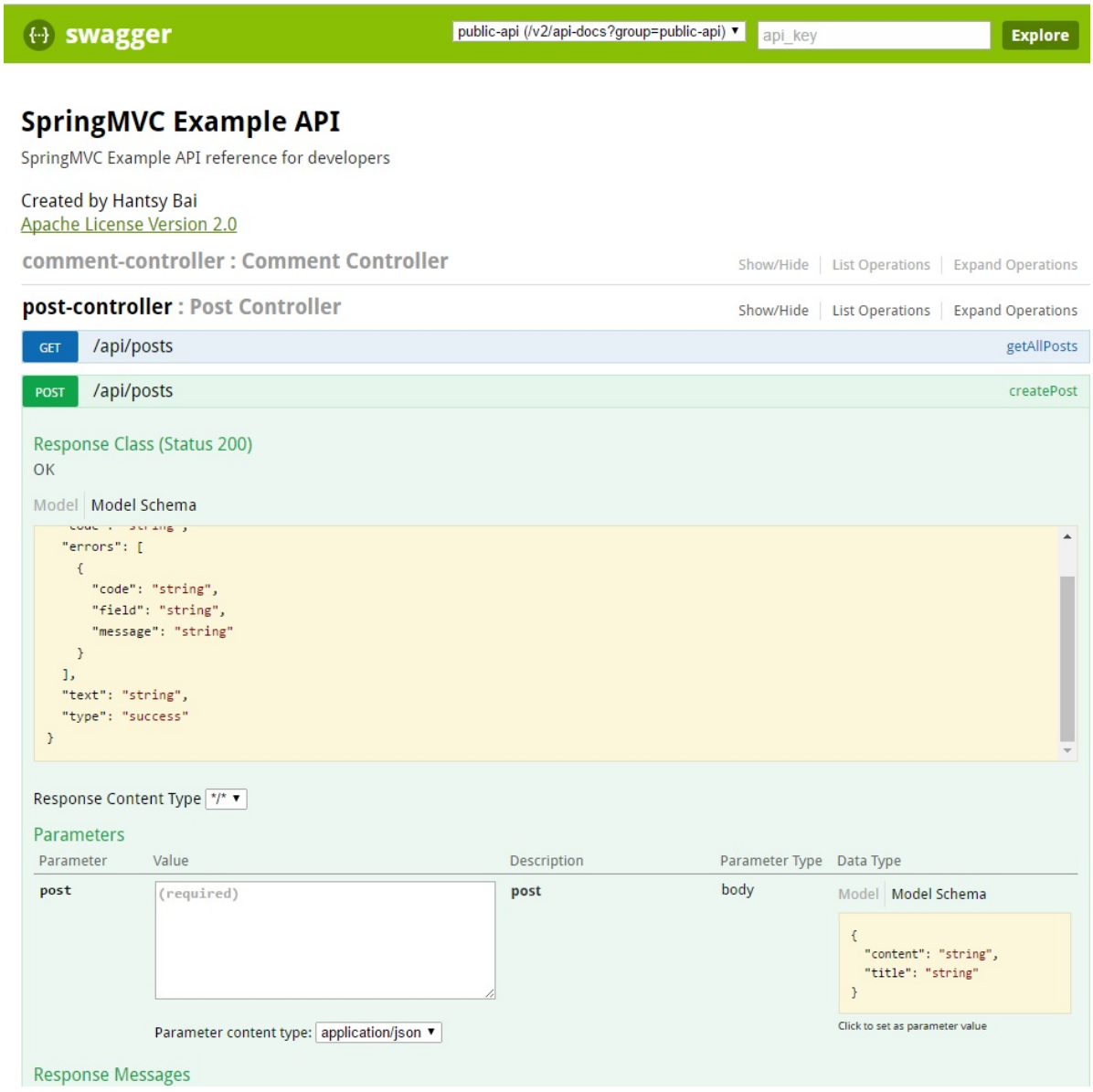
3. View REST APIs in swagger ui.

   Starts up this application via command line.

```
mvn tomcat7:run //or mvn spring-boot:run
```

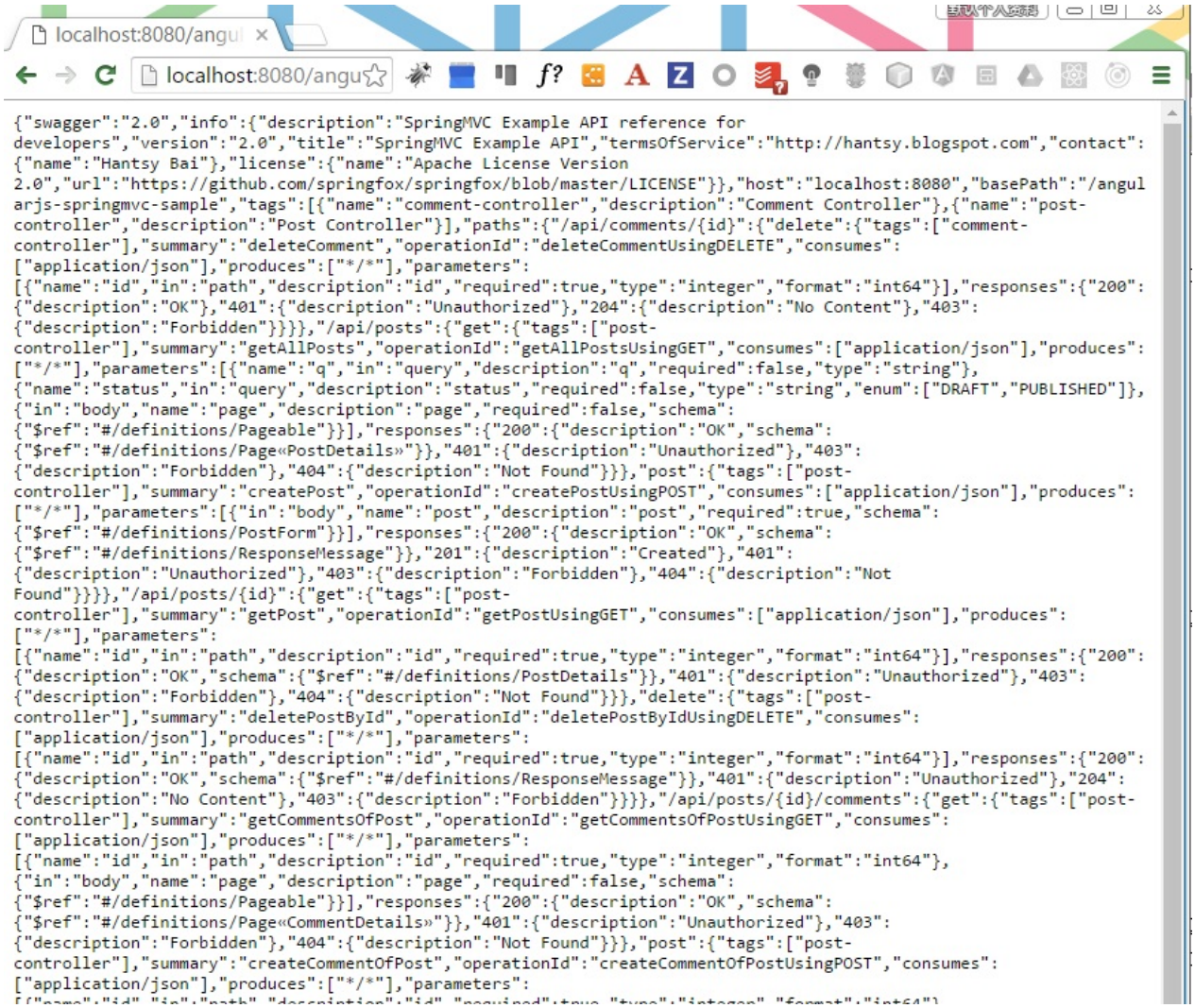
   Open browser and navigate http://localhost:8080/angularjs-springmvc-sample/swagger-ui.html.

   You will see the screen like the following.



# Documents REST APIs

In the above steps, the Swagger schema definition is generated at runtime, you can get the content via link:http://localhost:8080/angularjs-springmvc-sample/v2/api-docs?group=public-api. You will see the complete Swagger schema definition.




You can save this page content as a json file and upload to http://editor.swagger.io and edit it online.

The Swagger schema definition generation will consume lots of system resourcs at runtime.

Combined with Springfox, Swagger2Markup project, and Spring RestDocs, the Swagger schema definition can be converted to asciidocs, and with `asciidoctor-maven-plugin`, the asciidocs can be generated into static HTML5 or PDF files.

1. Add `swagger2-markup-maven-plugin` into *pom.xml* file.

```
 <!-- First, use the swagger2markup plugin to generate ascii
doc -->
 <plugin>
     <groupId>io.github.swagger2markup</groupId>
     <artifactId>swagger2markup-maven-plugin</artifactId>
     <version>${swagger2markup.version}</version>
     <dependencies>
         <dependency>
             <groupId>io.github.swagger2markup</groupId>
             <artifactId>swagger2markup-import-files-ext</ar
tifactId>
             <version>${swagger2markup.version}</version>
         </dependency>
         <dependency>
             <groupId>io.github.swagger2markup</groupId>
             <artifactId>swagger2markup-spring-restdocs-ext<
/artifactId>
             <version>${swagger2markup.version}</version>
         </dependency>
     </dependencies>
     <configuration>
         <swaggerInput>${swagger.input}</swaggerInput>
         <outputDir>${generated.asciidoc.directory}</outputD
ir>
         <config>
             <swagger2markup.markupLanguage>ASCIIDOC</swagge
r2markup.markupLanguage>
             <swagger2markup.pathsGroupedBy>TAGS</swagger2ma
rkup.pathsGroupedBy>

             <swagger2markup.extensions.dynamicOverview.cont
entPath>${project.basedir}/src/docs/asciidoc/extensions/over
view</swagger2markup.extensions.dynamicOverview.contentPath>
             <swagger2markup.extensions.dynamicDefinitions.c
ontentPath>${project.basedir}/src/docs/asciidoc/extensions/d
efinitions</swagger2markup.extensions.dynamicDefinitions.con
tentPath>
             <swagger2markup.extensions.dynamicPaths.content
```

```
Path>${project.basedir}/src/docs/asciidoc/extensions/paths</
swagger2markup.extensions.dynamicPaths.contentPath>
            <swagger2markup.extensions.dynamicSecurity.cont
entPath>${project.basedir}src/docs/asciidoc/extensions/secur
ity/</swagger2markup.extensions.dynamicSecurity.contentPath>

            <swagger2markup.extensions.springRestDocs.snipp
etBaseUri>${swagger.snippetOutput.dir}</swagger2markup.exten
sions.springRestDocs.snippetBaseUri>
            <swagger2markup.extensions.springRestDocs.defau
ltSnippets>true</swagger2markup.extensions.springRestDocs.de
faultSnippets>
        </config>
    </configuration>
    <executions>
        <execution>
            <phase>test</phase>
            <goals>
                <goal>convertSwagger2markup</goal>
            </goals>
        </execution>
    </executions>
 </plugin>
```

The `convertSwagger2markup` goal will convert Swagger schema definition into asciidocs.

2. Add `asciidoctor-maven-plugin` into `pom.xml` file.

```
 <!-- Run the generated asciidoc through Asciidoctor to gene
rate
 other documentation types, such as PDFs or HTML5 -->
 <plugin>
    <groupId>org.asciidoctor</groupId>
    <artifactId>asciidoctor-maven-plugin</artifactId>
    <version>1.5.3</version>
    <!-- Include Asciidoctor PDF for pdf generation -->
    <dependencies>
        <dependency>
            <groupId>org.jruby</groupId>
```

```
            <artifactId>jruby-complete</artifactId>
            <version>${jruby.version}</version>
        </dependency>
        <dependency>
            <groupId>org.asciidoctor</groupId>
            <artifactId>asciidoctorj-pdf</artifactId>
            <version>${asciidoctorj-pdf.version}</version>
        </dependency>
    </dependencies>
    <!-- Configure generic document generation settings -->
    <configuration>
        <sourceDirectory>${asciidoctor.input.directory}</so
urceDirectory>
        <sourceDocumentName>index.adoc</sourceDocumentName>

        <sourceHighlighter>coderay</sourceHighlighter>
        <attributes>
            <doctype>book</doctype>
            <toc>left</toc>
            <toclevels>3</toclevels>
            <numbered></numbered>
            <hardbreaks></hardbreaks>
            <sectlinks></sectlinks>
            <sectanchors></sectanchors>
            <generated>${generated.asciidoc.directory}</gen
erated>
        </attributes>
    </configuration>
    <!-- Since each execution can only handle one backend,
run
    separate executions for each desired output type -->
    <executions>
        <execution>
            <id>output-html</id>
            <phase>test</phase>
            <goals>
                <goal>process-asciidoc</goal>
            </goals>
            <configuration>
                <backend>html5</backend>
```

```
                <outputDirectory>${asciidoctor.html.output.
directory}</outputDirectory>
            </configuration>
        </execution>

        <execution>
            <id>output-pdf</id>
            <phase>test</phase>
            <goals>
                <goal>process-asciidoc</goal>
            </goals>
            <configuration>
                <backend>pdf</backend>
                <outputDirectory>${asciidoctor.pdf.output.d
irectory}</outputDirectory>
            </configuration>
        </execution>
    </executions>
 </plugin>
```

`asciidoctor-maven-plugin` will generate the asciidocs into HTML5 and PDF files.

3. Add `spring-restdocs` support.

   `spring-restdocs` will generate the sample code snippets from test, which can be combined into the final docs.

   Add related dependencies into `pom.xml` file.

```
 <dependency>
     <groupId>io.github.swagger2markup</groupId>
     <artifactId>swagger2markup-spring-restdocs-ext</artifac
tId>
     <version>${swagger2markup.version}</version>
     <scope>test</scope>
 </dependency>

 <dependency>
     <groupId>org.springframework.restdocs</groupId>
     <artifactId>spring-restdocs-mockmvc</artifactId>
     <scope>test</scope>
 </dependency>
```

Write test codes to generate sample code snippets.

```
 @WebAppConfiguration
 @RunWith(SpringRunner.class)
 @SpringBootTest(classes = {Application.class, SwaggerConfig
.class})
 public class MockMvcApplicationTest {

     String outputDir = System.getProperty("io.springfox.sta
ticdocs.outputDir");
     String snippetsDir = System.getProperty("io.springfox.s
taticdocs.snippetsOutputDir");
     String asciidocOutputDir = System.getProperty("generate
d.asciidoc.directory");

     @Rule
     public final JUnitRestDocumentation restDocumentation =
 new JUnitRestDocumentation(System.getProperty("io.springfox
.staticdocs.snippetsOutputDir"));

     @Inject
     private WebApplicationContext context;

     @Inject
     private ObjectMapper objectMapper;
```

```
    @Inject
    private PostRepository postRepository;

    private MockMvc mockMvc;

    private Post savedIdentity;

    @Before
    public void setUp() {
        this.mockMvc = webAppContextSetup(this.context)
                .apply(documentationConfiguration(this.rest
Documentation))
                .alwaysDo(document("{method-name}",
                        preprocessRequest(prettyPrint()),
                        preprocessResponse(prettyPrint())))
                .build();

        savedIdentity = postRepository.save(newEntity());
    }

    @Test
    public void createSpringfoxSwaggerJson() throws Excepti
on {
        //String designFirstSwaggerLocation = Swagger2Marku
pTest.class.getResource("/swagger.yaml").getPath();

        MvcResult mvcResult = this.mockMvc.perform(get("/v2
/api-docs")
                .accept(MediaType.APPLICATION_JSON))
                .andDo(
                        SwaggerResultHandler.outputDirector
y(outputDir)
                        .build()
                )
                .andExpect(status().isOk())
                .andReturn();

        //String springfoxSwaggerJson = mvcResult.getRespon
se().getContentAsString();
```

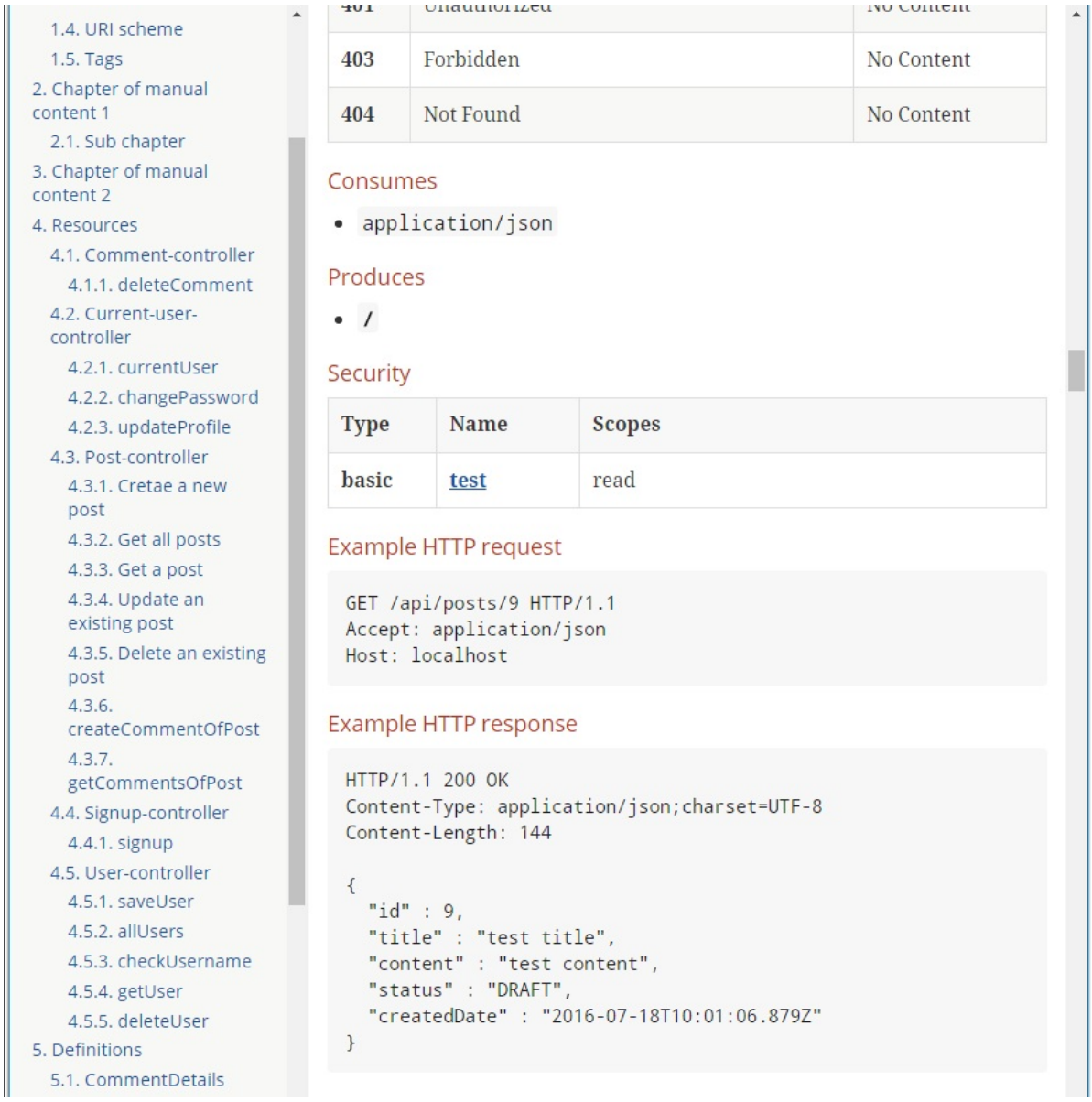
```
        //SwaggerAssertions.assertThat(Swagger20Parser.pars
e(springfoxSwaggerJson)).isEqualTo(designFirstSwaggerLocatio
n);
    }

    //    @Test
    //    public void convertToAsciiDoc() throws Exception
{
    //        this.mockMvc.perform(get("/v2/api-docs")
    //                .accept(MediaType.APPLICATION_JSON))
    //                .andDo(
    //                        Swagger2MarkupResultHandler.o
utputDirectory("src/docs/asciidoc")
    //                        .withExamples(snippetsDir).bu
ild())
    //                .andExpect(status().isOk());
    //    }

    @Test
    public void getAllPosts() throws Exception {
        this.mockMvc
                .perform(
                        get("/api/posts/{id}", savedIdentit
y.getId())
                        .accept(MediaType.APPLICATION_JSON)
                )
                //.andDo(document("get_a_post", preprocessR
esponse(prettyPrint())))
                .andExpect(status().isOk());
    }

    @Test
    public void getAllIdentities() throws Exception {
        this.mockMvc
                .perform(
                        get("/api/posts")
                        .accept(MediaType.ALL)
                )
                //.andDo(document("get_all_posts"))
                .andExpect(status().isOk());
```

```
    }

    @Test
    public void createPost() throws Exception {
        this.mockMvc
                .perform(
                        post("/api/posts")
                        .contentType(MediaType.APPLICATION_
JSON)
                        .content(newEntityAsJson())
                )
                //.andDo(document("create_a_new_post"))
                .andExpect(status().isCreated());
    }

    @Test
    public void updatePost() throws Exception {
        this.mockMvc
                .perform(
                        put("/api/posts/{id}", savedIdentit
y.getId())
                        .contentType(MediaType.APPLICATION_
JSON)
                        .content(newEntityAsJson())
                )
                //.andDo(document("update_an_existing_post"
))
                .andExpect(status().isNoContent());
    }

    @Test
    public void deletePost() throws Exception {
        this.mockMvc
                .perform(
                        delete("/api/posts/{id}", savedIden
tity.getId())
                        .contentType(MediaType.APPLICATION_
JSON)
                )
                //.andDo(document("delete_an_existing_post"
```

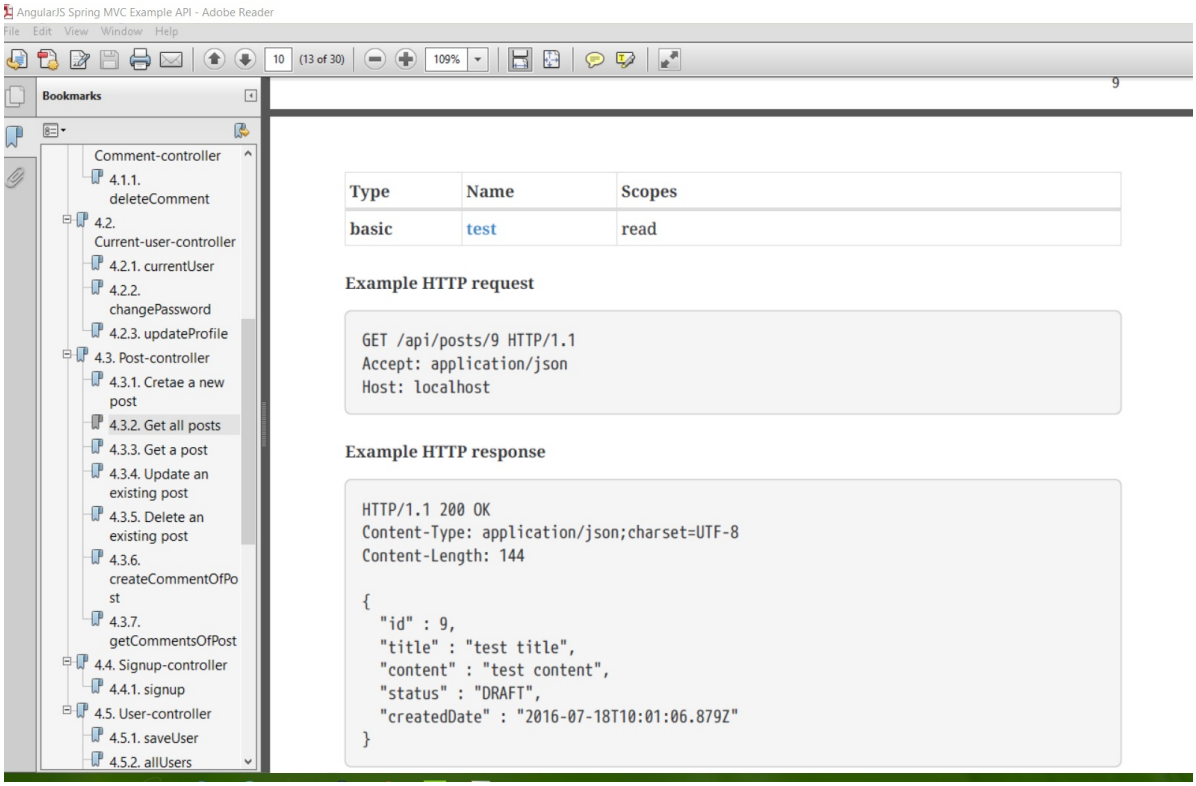
```
))
                .andExpect(status().isNoContent());
    }

    private Post newEntity() {
        Post post = new Post();
        post.setTitle("test title");
        post.setContent("test content");

        return post;
    }

    private String newEntityAsJson() throws JsonProcessingE
xception {
        return objectMapper.writeValueAsString(newEntity())
;
    }

 }
```

4. Run `mvn clean verify` to execute all tests and generate HTML5 and PDF file for the REST APIs.

   Open *\target\asciidoc\html\index.html* in browser, it looks like.




Open *\target\asciidoc\pdf\index.pdf* in Pdf viewer, it looks like.




# Source Code

Check out sample codes from my github account.

```
git clone https://github.com/hantsy/angularjs-springmvc-sample
```

Or the Spring Boot version:

```
git clone https://github.com/hantsy/angularjs-springmvc-sample-boot
```

Read the live version of thess posts from Gitbook:Building RESTful APIs with Spring MVC.

# Secures APIs

We have configured Spring Security in before posts.

In this post, I will show you using Spring Security to protect APIs, aka provides Anthentication and Anthorization service for this sample application.

- **Authentication** answers the question: if the user is a valid user.
- **Authorization** resolves the problem: if the authenticated user has corresponding permissions to access resources.

## Authentication

In Spring security, it is easy to configure JAAS compatible authentication strategy, such as FORM, BASIC, X509 Certiciate etc.

Unlike JAAS in which the authentication management is very dependent on the container itself. Spring Security provides some extension points(such as `UserDetails`, `UserDetailsService`, `Authority`) and allows developers to customize and implement the authentication and authorization in a progammatic approach.

Motioned in before posts, the simplest way to configure Spring security is using `AuthenticationManagerBuilder` to build essential required resources.

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
        throws Exception {
    auth.inMemoryAuthentication()
            .passwordEncoder(passwordEncoder())
            .withUser("admin").password("test123").authorities("
ROLE_ADMIN")
            .and()
                .withUser("test").password("test123").authoritie
s("ROLE_USER");
}
```

An in-memory database and a HTTP BASIC authentication is easy to prototype applications, as showing as above codes.

If you want to store users into your database, firstly create a custom `UserDetailsService` bean and implement the `findByUsername` method and return a `UserDetails` object.

```
public class SimpleUserDetailsServiceImpl implements UserDetails
Service {

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

    private UserRepository userRepository;

    public SimpleUserDetailsServiceImpl(UserRepository userRepos
itory) {
        this.userRepository = userRepository;
    }

    @Override
    public UserDetails loadUserByUsername(String username) throw
s UsernameNotFoundException {
        User user = userRepository.findByUsername(username);
        if (user == null) {
            throw new UsernameNotFoundException("username not fo
und:" + username);
        }

        log.debug("found by username @" + username);

        return user;

    }

}
```

`User` class implements `UserDetails`.

```
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
@Entity
@Table(name = "users")
public class User implements UserDetails, Serializable {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    @Id()
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Long id;

    @Column(name = "username")
    private String username;

    @Column(name = "password")
    private String password;

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

    @Column(name = "email")
    private String email;

    @Column(name = "role")
    private String role;

    @Column(name = "created_date")
    @CreatedDate
    private LocalDateTime createdDate;

    public String getName() {
        if (this.name == null || this.name.trim().length() == 0)
```

```
 {
            return this.username;
        }
        return name;
    }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities
() {
        return Arrays.asList(new SimpleGrantedAuthority("ROLE_"
+ this.role));
    }

    @Override
    public String getPassword() {
        return this.password;
    }

    @Override
    public String getUsername() {
        return this.username;
    }

    @Override
    public boolean isAccountNonExpired() {
        return true;
    }

    @Override
    public boolean isAccountNonLocked() {
        return true;
    }

    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }

    @Override
    public boolean isEnabled() {
```

```
        return true;
    }

}
```

Then configure `AuthenticationManager` with custom `UserDetailsService` instead of `inMemoryAuthentication`.

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
        throws Exception {
    auth.userDetailsService(new SimpleUserDetailsServiceImpl(use
rRepository))
        .passwordEncoder(passwordEncoder);
}
```

If you want to design a customized Authentication strategy, you could have to create a custom `AuthenticationEntryPoint` and `AuthenticationProvider` for it. We will discuss this later.

# Anthorization

Once user is authenticated, when he tries to access some resources, such as URL, or execute some methods, it should check if the resource is protected, or has granted permissions on executing the methods.

## Declarative URL pattern based authorizations

For REST APIs, the API resources are identified by URI, it is easy to grant authorizations via URL.

Override the `configure(HttpSecurity)` of `WebSecurityConfigurerAdapter`.

```
public class SecurityConfig extends WebSecurityConfigurerAdapter
 {

    @Override
    protected void configure(HttpSecurity http) throws Exception
 {
        http
            .authorizeRequests()
            .antMatchers("/api/ping")
            .permitAll()
        .and()
            .authorizeRequests()
            .antMatchers("/api/**")
            .authenticated()
            //....
    }
```

The access control is filter by the `Matcher`, there are two built-in matchers, Apache Ant path matcher, and perl like regex matchers. The later is a little complex, but more powerful.

`http...antMatchers("/api/**").authenticated()` means all resource URLs match '/api/**' need a valid authentication.

`http...antMatchers(HttpMethod.POST, "/api/posts").hasRoles("ADMIN")` indicates only users that have been granted **ADMIN** role have permission to create a new post.

Combined with resource URLs and HTTP methods, it follows rest convention exactly.

In the a real world application, you can centralize the *URL pattern*, *HTTP Method*, and granted *ROLES* into a certain persistent storage(such as RDBMS or NOSQL) and desgin a friendly web UI to control resource access.

## Method level authorizations

Like JAAS, Spring Security provides several annotations to authorize access on method level.

Firstly you should add `@EnableGlobalMethodSecurity` on `@Configuration` class to enable it.

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true, jsr250Enabled
 = true, securedEnabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter
 {

}
```

If **prePostEnabled** is true, the `@PreAuthorized` and `@PostAuthorized` can be used, it accept Spring EL string and evaluate the result.

For example, only *ADMIN* user can save post.

```
@PreAuthorized("hasRole('ADMIN')")
public void savePost(Post post){}
```

And only the post owner can update the post.

```
@PreAuthorized("#post.author.id==principal.id")
public void update(Post post){}
```

**jsr250Enabled** provides Java common anntotation compatibility, and allow you use JAAS annotations in Spring project.

**securedEnabled** enables the legacy `@Secured` annotation which does not accept Spring EL as property value.

```
@Secured("ROLE_USER")
public void savePost(Post post){}
```

## Programmatic authorizations

Spring provides APIs to fetch current principal info.

For example, get current Authetication from SecurityContextHolder.

```
Authentication auth = SecurityContextHolder.getContext().getAuth
entication();
```

And you can also inject current authenticated Principal like this.

```
publc List<Post> getByCurrentUser(@AuthenticationPrincipal Princ
ipal principal){}
```

After got the security principal info, you can control the authorizations in codes.

# Source Code

Check out sample codes from my github account.

```
git clone https://github.com/hantsy/angularjs-springmvc-sample
```

Or the Spring Boot version:

```
git clone https://github.com/hantsy/angularjs-springmvc-sample-b
oot
```

Read the live version of thess posts from Gitbook:Building RESTful APIs with Spring MVC.

# Upgrade to Spring Boot 1.4

Spring Boot 1.4 is a big jump, and introduced lots of new test facilities and aligned with the new technology stack, such as Spring framework 4.3 and Hibernate 5.2 and Spring Security 4.1, etc.

## Spring Boot 1.4

### New starter:spring-boot-starter-test

Spring Boot 1.4 brings a new starter for test scope, named `spring-boot-starter-test` .

Use the following:

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

Instead of:

```
<dependency>
    <groupId>com.jayway.jsonpath</groupId>
    <artifactId>json-path</artifactId>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.assertj</groupId>
    <artifactId>assertj-core</artifactId>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.hamcrest</groupId>
    <artifactId>hamcrest-core</artifactId>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <scope>test</scope>
</dependency>
```

`spring-boot-starter-test` includes the essential dependencies for test, such as json-path, assertj, hamcrest, mockito etc.

## New annotation: @SpringBootTest

Spring Boot 1.4 introduced a new annotation `@SpringBootTest` to unite the old `@IntegrationTest`, `@WebIntegrationTest`, `@SpringApplicationConfiguration` etc, in before versions.

A `webEnvironment` property of `@SpringBootTest` is use for deciding if set up a web environment for test.

There are some configuration options of the `webEnvironment`.

- **MOCK** is the default, provides a mock web environment.

- **NONE** does not give a web environment.
- **DEFINED_PORT** provides an embedded web environment and run the application on a defined port.
- **RANDOM_PORT** provides an embedded web environment, but use a random port number.

If **RANDOM_PORT** is used, add `@LocalSeverPort` annotation on an `int` field will inject the port number at runtime.

```
@LocalSeverPort
int port;
```

`@LocalServerPort` replaces the `@Value("${local.server.port}")` of Spring Boot 1.3.

Similarly, **classes** property is similar to the one of `@SpringApplicationConfiguration`. You can specify the configuration classes to be loaded for the test.

```
@SpringBootTest(classes = {Application.class, SwaggerConfig.clas
s})
```

The above code is equivalent to `@SpringApplicationConfiguration(classes={...})` in Spring Boot 1.3.

## New JUnit Runner: SpringRunner

Spring 1.4 introduced a new JUnit Runner, `SpringRunner`, which is an alias for the `SpringJUnit4ClassRunner`.

```
@RunWith(SpringRunner.class)
```

If you have to use other runners instead of `SpringRunner`, and want to use the Spring test context in the tests, declare a `SpringClassRule` and `SpringMethodRule` in the test to fill the gap.

```
@RunWith(AnotherRunner.class)
public class SomeTest{

    @ClassRule
    public static final SpringClassRule SPRING_CLASS_RULE = new
SpringClassRule();

    @Rule
    public final SpringMethodRule springMethodRule = new SpringM
ethodRule();

}
```

## Autoconfigure test slice

The most exciting feature provided in Spring Boot 1.4 is it provides capability to test some feature slice, which just pick up essential beans and configuration for the specific purpose based test.

Currently there is a series of new annotations available for this purpose.

**@JsonTest** provides a simple Jackson environment to test the json serialization and deserialization.

**@WebMvcTest** provides a mock web environment, it can specify the controller class for test and inject the `MockMvc` in the test.

```
@WebMvcTest(PostController.class)
public class PostControllerMvcTest{

    @Inject MockMvc mockMvc;

}
```

**@DataJpaTest** will prepare an embedded database and provides basic JPA environment for the test.

**@RestClientTest** provides REST client environment for the test, esp the RestTemplateBuilder etc.

These annotations are not composed with `SpringBootTest`, they are combined with a series of `AutoconfigureXXX` and a `@TypeExcludesFilter` annotations.

Have a look at `@DataJpaTest`.

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@BootstrapWith(SpringBootTestContextBootstrapper.class)
@OverrideAutoConfiguration(enabled = false)
@TypeExcludeFilters(DataJpaTypeExcludeFilter.class)
@Transactional
@AutoConfigureCache
@AutoConfigureDataJpa
@AutoConfigureTestDatabase
@AutoConfigureTestEntityManager
@ImportAutoConfiguration
public @interface DataJpaTest {}
```

You can add your `@AutoconfigureXXX` annotation to override the default config.

```
@AutoConfigureTestDatabase(replace=NONE)
@DataJpaTest
public class TestClass{
}
```

# JsonComponent

`@JsonComponent` is a specific `@Component` to register custome Jackson `JsonSerializer` and `JsonDeserializer`.

For example, custom `JsonSerializer` and `JsonDeserializer` are use for serializing and deserializing `LocalDateTime` instance.

```
@JsonComponent
@Slf4j
public class LocalDateTimeJsonComponent {

    public static class LocalDateTimeSerializer extends JsonSeri
alizer<LocalDateTime> {

        @Override
        public void serialize(LocalDateTime value, JsonGenerator
 jgen, SerializerProvider provider) throws IOException {
            jgen.writeString(value.atZone(ZoneId.systemDefault()
).toInstant().toString());
        }
    }

    public static class LocalDateTimeDeserializer extends JsonDe
serializer<LocalDateTime> {

        @Override
        public LocalDateTime deserialize(JsonParser p, Deseriali
zationContext ctxt) throws IOException, JsonProcessingException
{
            ObjectCodec codec = p.getCodec();
            JsonNode tree = codec.readTree(p);
            String dateTimeAsString = tree.textValue();
            log.debug("dateTimeString value @" + dateTimeAsStrin
g);
            return LocalDateTime.ofInstant(Instant.parse(dateTim
eAsString), ZoneId.systemDefault());
        }
    }
}
```

If you are using the Spring Boot default Jackson configuration, it will be activated by default when the application starts up.

But if you customized a `ObjectMapper` bean in your configuration, the autoconfiguration of `ObjectMapper` is disabled. You have to install `JsonComponentModule` manually, else the `@JsonComponent` beans will not be

scanned at all.

```
@Bean
public Jackson2ObjectMapperBuilder objectMapperBuilder(JsonCompo
nentModule jsonComponentModule) {

    Jackson2ObjectMapperBuilder builder = new Jackson2ObjectMapp
erBuilder();
    //...
            .modulesToInstall(jsonComponentModule);

    return builder;
}
```

## Mocking and spying Beans

Spring Boot 1.4 integrates Mockito tightly, and provides Spring specific `@MockBean` and `@MockSpy` annotations.

```
@RunWith(SpringRunner.class)
public class MockBeanTest {

    @MockBean
    private UserRepository userRepository;

}
```

## TestConfiguration and TestComponent

`TestConfiguration` and `TestComponent` are designated for test purpose, they are similar with `Configuration` and `Component`. Generic `Configuration` and `Component` can not be scanned by default in test.

```
public class TestClass{

    @TestConfiguration
    static class TestConfig{
    }

    @TestComponent
    static class TestBean{}

}
```

# Spring 4.3

There are a few features added in 4.3, the following is impressive.

## Composed annotations

The effort of Spring Composed are merged into Spring 4.3.

A series of new composed annotations are available, but the naming is a little different from Spring Composed.

For example, a RestController can be simplfied by the new annotations, list as the following table.

| Spring 4.2 | Spring 4.3 |
| --- | --- |
| @RequestMapping(value = "", method = RequestMethod.GET) | @GetMapping() |
| @RequestMapping(value = "", method = RequestMethod.POST) | @PostMapping() |
| @RequestMapping(value = "/{id}", method = RequestMethod.PUT) | @PutMapping(value = "/{id}") |
| @RequestMapping(value = "/{id}", method = RequestMethod.DELETE) | @DeleteMapping(value = "/{id}") |

A new `@RestControllerAdvice()` is provided for exception handling, it is combination of `@ControllerAdvice` and `@ResponseBody`. You can remove the `@ResponseBody` on the `@ExceptionHandler` method when use this new annotation.

For example, in the old Spring 4.2, an custom exception handler class looks like the following.

```
@ControllerAdvice()
public class RestExceptionHandler {

    @ExceptionHandler(value = {SomeException.class})
    @ResponseBody
    public ResponseEntity<ResponseMessage> handleGenericExceptio
n(SomeException ex, WebRequest request) {
    }
}
```

In Spring 4.3, it becomes:

```
@RestControllerAdvice()
public class RestExceptionHandler {

    @ExceptionHandler(value = {SomeException.class})
    public ResponseEntity<ResponseMessage> handleGenericExceptio
n(SomeException ex, WebRequest request) {
    }
}
```

## Auto constructor injection

If there is a only one constructor defined in the bean, the arguments as dependencies will be injected by default.

Before 4.3, you have to add `@Inject` or `@Autowired` on the constructor to inject the dependencies.

```
@RestController
@RequestMapping(value = Constants.URI_API_PREFIX + Constants.URI
_POSTS)
public class PostController {

    @Inject
    public PostController(BlogService blogService) {
        this.blogService = blogService;
    }
}
```

`@Inject` can be removed in Spring 4.3.

```
@RestController
@RequestMapping(value = Constants.URI_API_PREFIX + Constants.URI
_POSTS)
public class PostController {

    public PostController(BlogService blogService) {
        this.blogService = blogService;
    }
}
```

# Spring Security 4.1

The Java configuration is improved.

Before 4.1, you can configure `passwordEncoder` and `userDetailsService` via `AuthenticationManagerBuilder`.

```
@Configuration
@Order(SecurityProperties.ACCESS_OVERRIDE_ORDER)
protected static class ApplicationSecurity extends WebSecurityCo
nfigurerAdapter {

  @Override
  protected void configure(HttpSecurity http) throws Exception {
}

  @Override
  protected void configure(AuthenticationManagerBuilder auth)
          throws Exception {
      auth
          .userDetailsService(new SimpleUserDetailsServiceImpl(u
serRepository))
          .passwordEncoder(passwordEncoder);
  }

  @Bean
  @Override
  public AuthenticationManager authenticationManagerBean() throw
s Exception {
      return super.authenticationManagerBean();
  }

}
```

In 4.1, `userDetailsService` and `passwordEncoder` bean can be detected automaticially. No need to wire them by `AuthenticationManagerBuilder` manually. No need to override the `WebSecurityConfigurerAdapter` class and provide a custom configuration, a generic `WebSecurityConfigurerAdapter` bean is enough.

```
@Bean
public BCryptPasswordEncoder passwordEncoder() {
    BCryptPasswordEncoder passwordEncoder = new BCryptPasswordEn
coder();
    return passwordEncoder;
}

@Bean
public UserDetailsService userDetailsService(UserRepository user
Repository){
    return new SimpleUserDetailsServiceImpl(userRepository);
}

@Bean
public WebSecurityConfigurerAdapter securityConfig(){
    return new WebSecurityConfigurerAdapter() {
        @Override
        protected void configure(HttpSecurity http) throws Excep
tion {//...}
}
```

More details can be found in the What's New in Spring Security 4.1 chapter of Spring Secuirty documentation.

# Hibernate 5.2

The biggest change of Hibernate 5.2 is the packages had been reorganised, Hibernate 5.2 is Java 8 ready now.

**hibernate-java8** (Java 8 DateTime support) and **hibernate-entitymanager** (JPA provider bridge) are merged into **hibernate-core**.

Remove the following dependencies when upgrade to Hibernate 5.2.

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-java8</artifactId>
    <version>${hibernate.version}</version>
</dependency>
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-entitymanager</artifactId>
    <version>${hibernate.version}</version>
</dependency>
```

**NOTE**:If you are using Spring 4.2 with Hibernate 5.2.0.Final, it could break some dependencis, such as `spring-orm` , `spring-boot-data-jpa-starter` which depends on **hibernate-entitymanager**. Spring Boot 1.4.0.RC1 and Spring 4.3 GA fixed the issues. But I noticed in the Hibernate 5.2.1.Final, **hibernate-entitymanager** is back.

Hibernate 5.2 also added Java Stream APIs support, I hope it will be available in the next JPA specification.

# Source code

Clone the codes from Github account.

```
git clone https://github.com/hantsy/angularjs-springmvc-sample-boot
```