Contents

[Best Practice of Spring 1](#_Toc440972805)

[Work flow: 1](#_Toc440972806)

[**Spring boot** 3](#_Toc440972807)

[**features** 3](#_Toc440972808)

[Externalized configuration 3](#_Toc440972809)

[Spring MVC auto-configuration 4](#_Toc440972810)

[Spring security: 4](#_Toc440972811)

[Spring data 5](#_Toc440972812)

[Spring Data – MongoDB (http://projects.spring.io/spring-data/ ) 6](#_Toc440972813)

[Spring Data REST 9](#_Toc440972814)

[Spring Session 11](#_Toc440972815)

[<Spring in action> 12](#_Toc440972816)

[Spring基本概念 12](#_Toc440972817)

[Spring依赖注入 12](#_Toc440972818)

[Spring MVC 15](#_Toc440972819)

[**Sprint RESTful** 20](#_Toc440972820)

[Spring Security 21](#_Toc440972821)

[**Spring Session** 22](#_Toc440972822)

[**Spring test** 23](#_Toc440972823)

[**Spring database** 27](#_Toc440972824)

[**Spring – consume rest service** 30](#_Toc440972825)

[**Spring -- Scheduling Tasks** 31](#_Toc440972826)

[**Spring -- detect device** 31](#_Toc440972827)

[**spring – SSL** 32](#_Toc440972828)

[**spring – 外部化配置** 36](#_Toc440972829)

[**spring – Authentication** 37](#_Toc440972830)

[**Spring data rest** 45](#_Toc440972831)

[**Spring Data – JPA (relational database)** 48](#_Toc440972832)

# Best Practice of Spring

## Work flow:

1. > mkdir gs-rest-service/src/main/java
2. 在gs-rest-service项目下创建build.gradle

//设置脚本的运行环境, 借用maven依赖库管理, 依赖包的定义

//Spring Boot gradle plugin: all jars on the classpath -> “\*application-name.jar”; search public static void main() as runnable class；提供dependecies的默认版本（当然，依赖包可以指定版本覆盖）

buildscript {

repositories {

mavenCentral()

}

dependencies {

classpath("org.springframework.boot:spring-boot-gradle-plugin:1.3.1.RELEASE")

}

}

//声明构建项目类型

//指定项目为java项目，项目编译(在项目提示符下执行：gradle build)时生成项目的jar包。

apply plugin: 'java'

// java项目的eclipse开发环境构建.生成所需要的.project,.classpath等文件。

apply plugin: 'eclipse'

apply plugin: 'idea'

apply plugin: 'spring-boot'

//指定jar包名字:gs-rest-service-0.1.0.jar

jar {

baseName = 'gs-rest-service'

version = '0.1.0'

}

//指定仓库使用

repositories {

mavenCentral()

}

sourceCompatibility = 1.7

targetCompatibility = 1.7

//项目依赖定义，compile为编译级别依赖，还有testCompile为测试级别的依赖等， you don’t have to supply the version number if you are using the Spring Boot Gradle plugin

//”group:name:version”

dependencies {

compile("org.springframework.boot:spring-boot-starter-web")

testCompile("junit:junit")

}

task wrapper(type: Wrapper) {

gradleVersion = '2.3'

}

1. 导入项目到IDE STS：Import project directory as gradle project, set source path

Right click -> Properties -> Java Build Path -> Source -> Add Folder: src/main/java

**Spring boot**

**features**

@Controller: handle incoming web requests

@RestController: render the resulting string directly back to the caller besides @Controller’s function

@RequestMapping: provide “routing” information

@EnableAutoConfiguration: automatically configure spring application based on the jar dependencies that you have added. For example, if HSQLDB is on you classpath, and you have not manually configured and database connection beans, then in-memory databased will be auto-configured. If you add your DataSource bean, the default embedded database support will back away.

@ComponentScan: automatically pick up all application components, by default, application class is located in a root package, then all application components like @Configuration, @Component, @Service, @Repository, @Controller etc will be automatically registered as Spring Beans

@SpringBootApplication =

@Configuration +

@EnableAutoConfiguration +

@ComponentScan

@Autowired:

@ControllerAdvice: customize the JSON document to return for a particular controller and/or exception type.

@EnableGlobalMethodSecurity: add method-level security to a web application

@EnableWebSecurity: switch off the default web security configuration

### Externalized configuration

properties files, YAML files, environment variables, command-line arguments

1. command line option arguments -> property (add it to the spring environment

>–-spring.profiles.active=dev //一般作为最终运行的profiles

b. SpringApplication.setAdditionalProfiles(“dev”) //一般作为默认profiles

c. application.yml

Spring:

profiles:

active: dev

从而application-dev.yml or application-prod.yml is used

至于外部配置值 -> object，请见Spring.doc

### Spring MVC auto-configuration

Serve static resources: by defaults,

/static

/public

/resources

/META-INF/resources

src/main/webapp only works with war packaging

HttpMessageConverters: convert HTTP requests and responses. By defaults, objects can be converted to JSON (using the Jackson library) or XML (using the Jackon XML extension if available)

Template engines: by defaults, src/main/resources/templates/

Thymeleaf, Mustache, Groovy, FreeMaker, Velocity

Error Handling:

/error: map by default, handles all errors.

For machine clients it will produce a JSON response with details of the error, HTTP status and exception message.

For bowser clients there is a ‘whitelabel’ error view that renders the same data in HTML format (to customize it just add a view that resolves to ‘error’)

Controller advice to translate the server side exceptions to client-friendly json structures.

@ControllerAdvice

public class ExceptionTranslator {

@ExceptionHandler(ConcurrencyFailureException.class)

@ResponseStatus(HttpStatus.CONFLICT)

@ResponseBody

public ErrorDTO processConcurencyError(ConcurrencyFailureException ex) {

return new ErrorDTO(ErrorConstants.ERR\_CONCURRENCY\_FAILURE);

}

}

### Spring security:

If spring security is on the classpath then web applications will be secure by default with ‘basic’ authentication on all HTTP endpoints, the default AuthenticationManager has a single user (‘user’ username and random password, printed at INFO level when the application starts up), to change username and password as follows:

application.yml:

security:

user:

name: admin

password: secret

@Configuration

@EnableWebSecurity

@EnableGlobalMethodSecurity(prePostEnabled = true, securedEnabled = true)

public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

@Inject

private UserDetailsService userDetailsService;

@Inject

private RememberMeServices rememberMeServices;

@Bean

public PasswordEncoder passwordEncoder() {

return new BCryptPasswordEncoder();

}

//

@Inject

public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {

auth

.userDetailsService(userDetailsService)

.passwordEncoder(passwordEncoder());

}

@Override

public void configure(WebSecurity web) throws Exception {

web.ignoring()

.antMatchers("/scripts/\*\*/\*.{js,html}") ;

}

@Override

protected void configure(HttpSecurity http) throws Exception {

}

}

### Spring data

Caching: EhCache

Messaging: RabbitMQ

AMQP (Advanced Message Queuing Protocol) is a platform-neutral, wire-level protocol for message-oriented middleware.

RabbitMQ is a lightweight, reliable, scalable and portable message broker based on the AMQP protocol.

application.yml:

spring:

rabbitmq:

host: localhost

port: 5672

username: admin

password: secret

spring session

Test: unit test, integration test

Sprint boot actuator:

dependencies { compile “spring-boot-starter-actuator”}

Endpoints: /actuator, /autoconfig, /health, /metrics, /auditing/, …

application.properties:

security.user.name= admin 访问endpoints的权限

security.user.password= secret

management.security.role= SUPERUSER

management.context-path= /manage /health -> /manage/health

## Spring Data – MongoDB (<http://projects.spring.io/spring-data/> )

MongoDb[2] is an open-source NoSQL document database that uses a JSON-like schema instead of traditional table-based relational data.

Spring Data MongoDB[3] brings MongoDB to Spring applications in three ways:

* Annotations for object-to-document mapping
* Template-based database access with MongoTemplate
* Automatic runtime repository generation

In our back-server, spring data MongoDB is applied

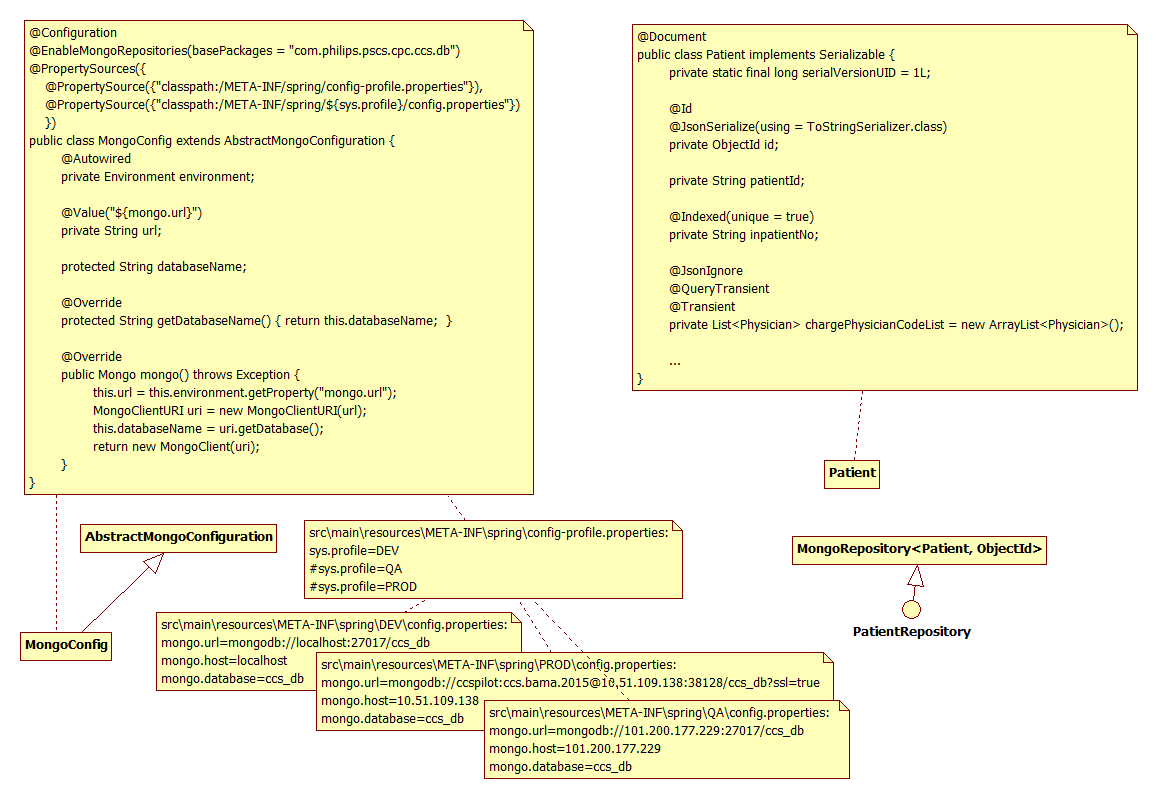


Figure 6. the diagram of spring data -mongodb

MongoConfig set configuration like MongoDB ip, port and database name. if database name does not exist, Spring MongoDB will automatically create one.

Patient is document like entity of JPA, will be mapped into collection of database ccs\_db.Patient

PatientRepository extends MongoRepository, it transitively extends the Repository marker interface. Any interface that extends Reposiotry will have an implementation automatically generated at runtime. PatientRepository will be implemented to read and write data to a MongoDB document PatientRepository interface has two parameters. The first is the type of @Document-annotated object that this repository deals with. The second is the type of the @Id-annotated property.

There are 3 common ways of defining query methods:

* Spring Data automatically generate 18 convenient methods [3]
* Spring Data supports a method-naming convention that helps Spring Data to automatically generate implementations for the methods that follow that convention.

For example, nothing else needs to be done to implement API: Patient findByPatientId(String patientId). When creating the repository implementation, Spring Data will examine any methods in the repository interface, parse the method name, and attempt to understand the method’s purpose in the context of the persisted object. In essence, Spring Data defines a sort of miniature domain-specific language (DSL) where persistence details are expressed in repository method signatures.

* Declare custom queries

You can use the @Query annotation to provide Spring Data with the query that should be performed. You still don’t write the implementation of the findPatientByPhilipsMail() method. You only give the query, hinting to Spring Data about how it should implement the method.

**public** **interface** PatientRepository **extends** MongoRepository<Patient, ObjectId>{

Patient findByInpatientNo(String inpatientNo);

@Query(“select p from Patient p where p.email like ‘%philips.com’ “)

List<Patient> findPatientsByPhilipsMail() ;

}

## Spring Data REST

**(**[**http://projects.spring.io/spring-data-rest/**](http://projects.spring.io/spring-data-rest/) **)**

Spring Data REST builds on top of **Spring Data repositories**, analyzes your application's **domain model** and **exposes hypermedia-driven HTTP resources** for aggregates contained in the model.

Via hypermedia-driven interface, there is no need to exchange a formal contract or interface document with your customers. **A hypermedia-driven site provides information to navigate the site's REST interfaces dynamically by including hypermedia links with the responses**. This capability differs from that of SOA-based systems and WSDL-driven interfaces. **With SOA, servers and clients usually must access a fixed specification** that might be staged somewhere else on the website, on another website, or perhaps distributed by email.

1. >mongod –dbpath ./data //启动mongo db
2. build.gradle

compile("org.springframework.boot:spring-boot-starter-data-rest")

compile("org.springframework.boot:spring-boot-starter-data-mongodb")

compile 'org.springframework.data:spring-data-rest-hal-browser'

1. create a domain object

src/main/java/com.philips.rest.domain.Person.java

public class Person {

@Id private String id ;

private String firstName ;

private String lastName ;

private String age ;

}

1. create repository

src/main/java/com.philips.rest.db.PersonRepository.java

public interface PersonRepository extends MongoRepository<Person, String>{

List<Person> findByLastName(@Param("name") String name) ;

}

1. run

e.1 expose RESTful API

|  |  |  |
| --- | --- | --- |
| Path | Method | Header |
| /persons{?page,size,sort}  (等于默认：?page=0&size=20) | GET |  |
| /persons/  {  “firstName”: “QiZhong”,  “lastName”: “Lin”,  “age”: 36  } | POST | Content-Type: application/json |
| /persons/{id} | GET |  |
| /persons/{id}  {  “firstName”: “QiZhong Jr.”,  “lastName”: “Lin”,  “age”: 3  } | PUT | Content-Type: application/json |
| /persons/{id}  {  “firstName”: “QiZhong Jr.”  } | PATCH | Content-Type: application/json |
| /persons/{id} | DELETE |  |
| /persons/search/findByLastName?name={name} | GET |  |

e.2 output HAL(hypertext application language) format

including resource link (from root URL to deep), **tools: HAL Browser**

localhost:8080/

## Spring Session

**(**[**http://projects.spring.io/spring-session/**](http://projects.spring.io/spring-session/) **)**

第一次login时，生成session id,同时在redis里以session id 为key,保存用户的信息；第二次就不需要login了，因为服务器与用户（浏览器）保存通话状态，服务端每次会发sessionid,浏览器会带上sessionid请求。

同浏览器，不论是同一tab,不同tab,或再启动一个浏览器，同一个session id有效，都会保存通知状态

//浏览器访问，sessionid采用默认的cookies传送，session信息存入redis数据库

src/main/java/session/config/HttpSessionConfig.java

@EnableRedisHttpSession

**public** **class** HttpSessionConfig {}

//restful 访问，sessionid采用httpheader传送，session信息存入redis数据库

第一次访问需要用户名和密码，服务端返回x-auth-token于httpheader

$curl –v localhost:8080 –u user:password

第二次用x-auth-token访问即可，不需要用户名和密码

$curl –v localhost:8080 –H “x-auth-token：…”

退出使token无效

$curl –v localhost:8080/logout –H “x-auth-token:…”

重复之前的过程

@EnableRedisHttpSession

**public** **class** HttpSessionConfig {

@Bean

**public** HttpSessionStrategy httpSessionStrategy() {

**return** **new** HeaderHttpSessionStrategy() ;

}

}

# <Spring in action>

## Spring基本概念

基本概念：DI（依赖注入），AOP（面向切面编程）, 模板

许多框架（如MFC）通过强迫应用继承它们的类或实现它们的接口从而来让应用跟框架绑死。Spring通过依赖注入

耦合具有两面性，一方面紧密耦合的代码难以测试，难以复用，难以理解，另一方面一定程度的耦合又是必须的，为了完成有实际意义的功能，不同的类必须以适当的方式进行交互

依赖注入：对象的依赖关系将由负责协调系统中各个对象的第三方组件在创建对象时设定。对象无需自行创建或管理它们的依赖关系，依赖关系将被自动注入到需要它们的对象中去。Spring通过应用上下文（Application Context）装载Bean的定义并把它们组装起来。Spring应用上下文全权负责对象的创建和组装。

装配(wiring):创建应用对象之间协作关系的行为。

Spring是一个基于容器的框架。我们需要配置Spring来告诉容器它需要加载哪里些Bean和如何装配这些Bean,这样才能确保它们能够彼此协作。

每个Spring容器即ApplicationContext，分配一个Bean时，它总是返回Bean的同一个实例。

## Spring依赖注入

1. XML-config方式

src/main/resources/META-INF/spring/knights.xml (定义bean)

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"*>

<bean id=*"knight"* class=*"com.springinaction.knights.BraveKnight"*>

<constructor-arg ref=*"quest"* />

</bean>

<bean id=*"quest"*

class=*"com.springinaction.knights.SlayDragonQuest"* />

</beans>

src/main/knightMain.java (引用bean)

**public** **class** KnightMain {

**public** **static** **void** main(String[] args) {

ApplicationContext context =

**new** ClassPathXmlApplicationContext("META-INF/spring/knights.xml");

Knight knight = (Knight) context.getBean("knight");

knight.embarkOnQuest();

}

}

1. Java-config 方式

src/main/java/knightConfg.java

@Configuration

**public** **class** KnightConfig {

@Bean

**public** Knight knight() {

**return** **new** BraveKnight(quest());

}

@Bean

**public** Quest quest() {

**return** **new** SlayDragonQuest(System.***out***);

}

}

src/main/knightMain.java (引用bean)

**public** **class** KnightMain {

**public** **static** **void** main(String[] args) {

ApplicationContext context =

**new** AnnotationConfigApplicationContext(com.philips.KnightConfig.class);

Knight knight = (Knight) context.getBean("knight");

knight.embarkOnQuest();

}

}

1. **auto-detect and auto-wire (highly importance)**

auto-detect(自动检测ComponentScan)：让Spring自动识别哪些类需要被配置成Spring Bean,从而减少对<bean>元素的使用

ComponentScan扫描config所在的包及其所有子包，查找能够自动注册为Spring Bean的类。这些类有如下注解：@Component, @Controller, @Repository, @Service

autowire(自动装配): 消除配置Bean的属性和构造器参数，让Spring自动识别如何装配Bean的依赖关系

src/main/java/domain/CDPlayer.java

@Component

**public** **class** CDPlayer **implements** MediaPlayer {

**private** CompactDisc cd;

@Autowired

**public** CDPlayer(CompactDisc cd) {

**this**.cd = cd;

}

**public** **void** play() {

cd.play();

}

}

src/main/java/domain/SgtPeppers.java

@Component

**public** **class** SgtPeppers **implements** CompactDisc {

**private** String title = "Sgt. Pepper's Lonely Hearts Club Band";

**private** String artist = "The Beatles";

**public** **void** play() {

System.***out***.println("Playing " + title + " by " + artist);

}

}

src/main/java/Config.java

@Configuration

@ComponentScan

**public** **class** Config {}

src/test/java/Test.java

//自动创建Spring ApplicationContext

@RunWith(SpringJUnit4ClassRunner.**class**)

//导入Config配置文件

@ContextConfiguration(classes=Config.**class**)

**public** **class** Test {

@Autowired

**private** MediaPlayer player;

@Autowired

**private** CompactDisc cd;

@Test

**public** **void** cdShouldNotBeNull() {

*assertNotNull*(cd);

}

@Test

**public** **void** play() {

player.play();

}

}

## Spring MVC



1. XML-config

src/main/webapp/WEB-INF/web.xml

<web-app version=*"2.5"* xmlns=*"http://java.sun.com/xml/ns/javaee"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"*>

//除了DispatcherServlet所需的mvc配置外，还有别的配置如领域层(bean)，服务层(bean)，持久层，数据源，甚至web-flow配置，都放入servletname-servlet.xml也可以，但组织到多个文件比较清晰。

// ContextLoaderListener是一个Servlet监听器，会加载其他的配置文件到一个Spring应用上下文中

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/spring/domain.xml

/WEB-INF/spring/services.xml

/WEB-INF/spring/dataaccess.xml

/WEB-INF/spring/flow.xml</param-value>

</context-param>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

//tomcat环境--实例化前端控制器Sevlet(DispatcherServlet)，创建Spring应用上下文。同时基于约定优先配置，DispatcherServlet加载servletname-servlet.xml到Spring应用上下文

<servlet>

<servlet-name>SpringPizza</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value></param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>SpringPizza</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

</web-app>

src/main/webapp/WEB-INF/ SpringPizza-Sevlet.xml

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:mvc="http://www.springframework.org/schema/mvc">

<bean id="multipartResolver" class=

"org.springframework.web.multipart.commons.CommonsMultipartResolver"

p:maxUploadSize="500000" />

//自动扫描组件，并注册为bean

<context:component-scan base-package="com.habuma.spitter.mvc" />

// 默认情况下，DispatcherServlet创建并使用BeanNameUrlHandlerMapping and DefaultAnnotationHandlerMapping （处理器映射，将请求分发给哪个控制器），将请求映射给使用@RequestMapping注解的控制器和控制器方法

//mvc:annotation-driven将请求参数绑定到控制器的方法参数，校验支持，信息转换以及对域格式化的支持

<mvc:annotation-driven/>

//DispatcherServlet配置视图解析器

<bean class=”org.springframework.web.servlet.view.InternalResourceViewResolver”>

<property name=”prefix” value=”/WEB-INF/views/”/>

<property name=”suffix” value=”.jsp” />

</bean>

//配置处理静态资源的请求

<mvc:resources mapping="/resources/\*\*" location="/resources/" />

</beans>

可以组织以上的xml配置文件如下：

src/main/webapp/WEB-INF/web.xml

<web-app version=*"2.5"* xmlns=*"http://java.sun.com/xml/ns/javaee"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"*>

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/spring/root-config.xml</param-value>

</context-param>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

<servlet>

<servlet-name>SpringPizza</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value></param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>SpringPizza</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

</web-app>

src/main/webapp/WEB-INF/spring/root-config.xml

<beans xmlns=*"http://www.springframework.org/schema/beans"*>

<import resource=*"mvc.xml"* />

<import resource=*"flow.xml"* />

<import resource=*"services.xml"* />

<import resource=*"domain.xml"* />

<import resource=*"dataaccess.xml"* />

<context:component-scan base-package=*"com.springinaction.pizza"* />

</beans>

//将servletname-servlet.xml 配置内容放入mvc.xml中

src/main/webapp/WEB-INF/spring/mvc.xml

src/main/webapp/WEB-INF/spring/domain.xml

src/main/webapp/WEB-INF/spring/service.xml

src/main/webapp/WEB-INF/spring/dataaccess.xml

src/main/webapp/WEB-INF/spring/flow.xml

1. **Java-config (highly importance)**

src/main/java/spittr/config/SpitterWebInitializer.java

//tomcat环境寻找到该类，从而会自动实例化DiapatcherServlet和Spring应用上下文

**public** **class** SpitterWebInitializer **extends** AbstractAnnotationConfigDispatcherServletInitializer {

//导入beans,等效于ContextLoaderListeners监听器加载beans于Spring应用上下文中

@Override

**protected** Class<?>[] getRootConfigClasses() {

**return** **new** Class<?>[] { RootConfig.**class** };

}

//导入beans,等效于DispatcherServlet加载servletname-servlet.xml

@Override

**protected** Class<?>[] getServletConfigClasses() {

**return** **new** Class<?>[] {spittr.web.WebConfig.**class** };

}

@Override

**protected** String[] getServletMappings() {

**return** **new** String[] { "/" };

}

}

src/main/java/spittr/web/WebConfig.java

@Configuration

@EnableWebMvc //等效于mvc:annotation-driven

@ComponentScan("spittr.web") //自动扫描web包的controller,service,repository…作为bean

**public** **class** WebConfig **extends** WebMvcConfigurerAdapter {

//配置视图解析器

@Bean

**public** ViewResolver viewResolver() {

InternalResourceViewResolver resolver = **new** InternalResourceViewResolver();

resolver.setPrefix("/WEB-INF/views/");

resolver.setSuffix(".jsp");

**return** resolver;

}

//配置静态资源处理器

@Override

**public** **void** configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {

configurer.enable();

}

}

src/main/java/spittr/config/RootConfig.java

//自动扫描spittr包及其子包，加载bean和\*\*Config下的Bean

@Configuration

@Import(DataConfig.**class**)

@ComponentScan(basePackages={"spittr"},

excludeFilters={

@Filter(type=FilterType.***CUSTOM***, value=WebPackage.**class**)

})

**public** **class** RootConfig {

**public** **static** **class** WebPackage **extends** RegexPatternTypeFilter {

**public** WebPackage() {

**super**(Pattern.*compile*("spittr\\.web"));

}

}

}

src/main/java/spittr/config/DataConfig.java

@Configuration

**public** **class** DataConfig {

@Bean

**public** DataSource dataSource() {

**return** **new** EmbeddedDatabaseBuilder()

.setType(EmbeddedDatabaseType.***H2***)

.addScript("schema.sql")

.build();

}

@Bean

**public** JdbcOperations jdbcTemplate(DataSource dataSource) {

**return** **new** JdbcTemplate(dataSource);

}

}

src/main/java/spittr/config/\*\*\*Config.java

**Sprint RESTful**

**Spring boot (highly importance)**

Src/main/java/Greeting.java

public class Greeting {

private final long id;

private final String content;

public Greeting(long id, String content) {

this.id = id;

this.content = content;

}

…

}

src/main/java/GreetingController.java

@RestController

public class GreetingController {

private static final String template = "Hello, %s!";

private final AtomicLong counter = new AtomicLong();

@RequestMapping("/greeting")

public Greeting greeting(@RequestParam(value="name", defaultValue="World") String name) {

return new Greeting(counter.incrementAndGet(),

String.format(template, name));

}

}

src/main/java/Application.java

@SpringBootApplication

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

@SpringBootApplication is a convenience annotation that adds all of the following:

***@Configuration*** tags the class as a source of bean definitions for the application context.

***@EnableAutoConfiguration*** tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings.

Normally you would add ***@EnableWebMvc*** for a Spring MVC app, but Spring Boot adds it automatically when it sees spring-webmvc on the classpath. This flags the application as a web application and activates key behaviors such as setting up a DispatcherServlet.

***@ComponentScan*** tells Spring to look for other components, configurations, and services in the the hello package, allowing it to find the HelloController.

## Spring Security

身份验证authentication和授权authorization

保护URL访问 -----Servlet filter

保护方法调用 -----Spring AOP

Spring boot方式，只要Spring security包在classpath里，basic authentication on all HTTP endpoints会默认生成。用户名是user,密码是tomcat启动是随机生成（见log信息）

一般情况下配置Security如下

@Configuration

@EnableWebMvcSecurity

**public** **class** SecurityConfig **extends** WebSecurityConfigurerAdapter {

@Override

**protected** **void** configure(HttpSecurity http) **throws** Exception {

http

//不采用默认login，用指定login页面

.formLogin().loginPage("/login").and()

//login out

.logout().logoutSuccessUrl("/").and()

// remember me

.rememberMe().tokenRepository(**new** InMemoryTokenRepositoryImpl())

.tokenValiditySeconds(2419200).key("spittrKey").and()

//http 基本认证

.httpBasic().realmName("Spittr").and()

//授权请求

.authorizeRequests()

.antMatchers("/").authenticated()

.antMatchers("/spitter/me").authenticated()

.antMatchers(HttpMethod.***POST***, "/spittles").authenticated()

.anyRequest().permitAll().and()

//加密请求（require HTTPS）

requireChannel().antMatchers(“/spitter/form”).requireSecure() ;

}

//生成用户信息，有四种方式：内存，关系型数据库，LDAP服务器，自定义

@Override

**protected** **void** configure(AuthenticationManagerBuilder auth) **throws** Exception {

//内存方式

auth.inMemoryAuthentication().withUser("user").password("password").roles("USER");

//关系型数据库

Auth.jdbcAuthentication().dataSource(dataSource) ;

//LDAP服务器

Auth.ldapAuthentication().userSearchFilter(“(uid={0})”).groupSearchFilter(“member={0}”).contextSource().url(“ldap://localhost:389/dc=philips,dc=com”) ;

}

}

**Spring Session**

第一次login时，生成session id,同时在redis里以session id 为key,保存用户的信息；第二次就不需要login了，因为服务器与用户（浏览器）保存通话状态，服务端每次会发sessionid,浏览器会带上sessionid请求。

同浏览器，不论是同一tab,不同tab,或再启动一个浏览器，同一个session id有效，都会保存通知状态

//浏览器访问，sessionid采用默认的cookies传送，session信息存入redis数据库

src/main/java/session/config/HttpSessionConfig.java

@EnableRedisHttpSession

**public** **class** HttpSessionConfig {}

//restful 访问，sessionid采用httpheader传送，session信息存入redis数据库

第一次访问需要用户名和密码，服务端返回x-auth-token于httpheader

$curl –v localhost:8080 –u user:password

第二次用x-auth-token访问即可，不需要用户名和密码

$curl –v localhost:8080 –H “x-auth-token：…”

退出使token无效

$curl –v localhost:8080/logout –H “x-auth-token:…”

重复之前的过程

@EnableRedisHttpSession

**public** **class** HttpSessionConfig {

@Bean

**public** HttpSessionStrategy httpSessionStrategy() {

**return** **new** HeaderHttpSessionStrategy() ;

}

}

**Spring test**

Mockito中 when(…).thenReturn(…) 这样的语法便是设置方法调用的返回值。另外也可以设置方法在何时调用会抛异常等。Mock对象用来验证测试中所依赖对象间的交互是否能够达到预期。 Mockito中用 verify(…).methodXxx(…) 语法来验证 methodXxx方法是否按照预期进行了调用

创建Mock对象: mock(class or interface)

设置方法调用的预期返回: 通过when(mock.someMethod()).thenReturn(value) 来设定mock对象某个方法调用时的返回值

验证方法调用:mock对象一旦建立便会自动记录自己的交互行为，所以我们可以有选择的对它的 交互行为进行验证。在Mockito中验证mock对象交互行为的方法是verify(mock).someMethod(…)

src/test/java/Test.java

//自动创建Spring ApplicationContext

@RunWith(SpringJUnit4ClassRunner.**class**)

//导入Config配置文件

@ContextConfiguration(classes=Config.**class**)

//@ContextConfiguration(classes={AppConfig.**class,**MongodbConfi**.class**},loader=AnnotationConfigContextLoader**.class**)

**public** **class** Test {

@Autowired

**private** MediaPlayer player;

@Autowired

**private** CompactDisc cd;

@Test

**public** **void** cdShouldNotBeNull() {

*assertNotNull*(cd);

}

@Test

**public** **void** knightShouldEmbarkOnQuest() {

Quest mockQuest = *mock*(Quest.**class**);

BraveKnight knight = **new** BraveKnight(mockQuest);

knight.embarkOnQuest();

*verify*(mockQuest, *times*(1)).embark();

}

}

MockMvc

**public** **class** SpittleControllerTest {

@Test

**public** **void** houldShowRecentSpittles() **throws** Exception {

//生成模拟数据

List<Spittle> expectedSpittles = createSpittleList(20);

//mock repositority

SpittleRepository mockRepository = *mock*(SpittleRepository.**class**);

*when*(mockRepository.findSpittles(Long.***MAX\_VALUE***, 20))

.thenReturn(expectedSpittles);

//mock spring mvc

SpittleController controller = **new** SpittleController(mockRepository);

MockMvc mockMvc = *standaloneSetup*(controller)

.setSingleView(**new** InternalResourceView("/WEB-INF/views/spittles.jsp"))

.build();

//assert expections

mockMvc.perform(*get*("/spittles"))

.andExpect(*view*().name("spittles"))

.andExpect(*model*().attributeExists("spittleList"))

.andExpect(*model*().attribute("spittleList",

*hasItems*(expectedSpittles.toArray())));

}

}

**Unit test for RESTful entry point**

**public** **class** PatientControllerTest {

**private** **static** MockMvc generateMockMvc(PatientRepository mockRepository){

PatientController controller = **new** PatientController() ;

controller.setPatientRepository(mockRepository);

MockMvc mockMvc = *standaloneSetup*(controller).build() ;

**return** mockMvc ;

}

@Test

**public** **void** shouldGetPatientList() **throws** Exception {

List<Patient> expectedPatients = PatientGenerator.generatePatients(2) ;

PatientRepository mockRepository = *mock*(PatientRepository.**class**) ;

*when*(mockRepository.findAll()).thenReturn(expectedPatients) ;

MockMvc mockMvc = PatientControllerTest.*generateMockMvc*(mockRepository) ;

MvcResult result = mockMvc.perform(*get*(Util.***BASE\_URL***+"/"))

.andExpect(*status*().isOk())

.andExpect(*content*().contentType(Util.***CONTENT\_TYPE***))

.andExpect(*jsonPath*("$.statusMessage").value("Success"))

.andExpect(*jsonPath*("$.patientList", *hasSize*(2)))

.andReturn() ;

String content = result.getResponse().getContentAsString() ;

System.***out***.println("response should have some patients:") ;

System.***out***.println(content) ;

}

@Test

@SuppressWarnings("unchecked")

**public** **void** dbFailure() **throws** Exception {

PatientRepository mockRepository = *mock*(PatientRepository.**class**) ;

*when*(mockRepository.findAll()).thenThrow(Exception.**class**) ;

MockMvc mockMvc = PatientControllerTest.*generateMockMvc*(mockRepository) ;

MvcResult result = mockMvc.perform(*get*(Util.***BASE\_URL***+"/"))

.andExpect(*status*().isOk())

.andExpect(*content*().contentType(Util.***CONTENT\_TYPE***))

.andExpect(*jsonPath*("$.statusMessage").value("Failure"))

.andExpect(*jsonPath*("$.patientList", *hasSize*(0)))

.andReturn() ;

String content = result.getResponse().getContentAsString() ;

System.***out***.println("response should have 0 patients:") ;

System.***out***.println(content) ;

}

}

**Integration test for RESTful entry point**

@RunWith(SpringJUnit4ClassRunner.**class**)

@SpringApplicationConfiguration(classes = CcsApplication.**class**)

@WebAppConfiguration

**public** **class** PatientControllerIntegrationTest {

**private** MockMvc mockMvc ;

@Autowired

**private** WebApplicationContext webApplicationContext;

@Autowired

**private** PatientRepository repository;

@Autowired

MongoOperations mongoOps;

@Before

**public** **void** setup() **throws** Exception {

**this**.mockMvc = *webAppContextSetup*(webApplicationContext).build();

}

@Test

**public** **void** testGetPatientList() **throws** Exception {

MvcResult result = mockMvc.perform(*get*(Util.***BASE\_URL*** + "/")

.contentType(Util.***CONTENT\_TYPE***))

.andExpect(*status*().isOk())

.andExpect(*content*().contentType(Util.***CONTENT\_TYPE***))

.andExpect(*jsonPath*("$.statusMessage").value("Success"))

.andExpect(*jsonPath*("$.patientList", *hasSize*(500)))

.andReturn() ;

String content = result.getResponse().getContentAsString() ;

System.***out***.println("response content:") ;

System.***out***.println(content) ;

}

@Test

**public** **void** testGetPatientListByPagination() **throws** Exception {

Pagination pagination = **new** Pagination() ;

pagination.setProperty("age");

pagination.setDirection("asc");

pagination.setPage(0);

pagination.setSize(3);

MvcResult result = mockMvc.perform(*post*(Util.***BASE\_URL***+"/byPagination")

.content(**this**.json(pagination))

.contentType(Util.***CONTENT\_TYPE***))

.andExpect(*status*().isOk())

.andExpect(*content*().contentType(Util.***CONTENT\_TYPE***))

.andExpect(*jsonPath*("$.statusMessage").value("Success"))

.andExpect(*jsonPath*("$.patientList", *hasSize*(3)))

.andReturn() ;

String content = result.getResponse().getContentAsString() ;

System.***out***.println("response content:") ;

System.***out***.println(content) ;

}

**private** HttpMessageConverter mappingJackson2HttpMessageConverter;

@Autowired

**void** setConverters(HttpMessageConverter<?>[] converters) {

**this**.mappingJackson2HttpMessageConverter = Arrays.*asList*(converters).stream().filter(

hmc -> hmc **instanceof** MappingJackson2HttpMessageConverter).findAny().get();

Assert.*assertNotNull*("the JSON message converter must not be null",

**this**.mappingJackson2HttpMessageConverter);

}

**protected** String json(Object o) **throws** IOException {

MockHttpOutputMessage mockHttpOutputMessage = **new** MockHttpOutputMessage();

**this**.mappingJackson2HttpMessageConverter.write(o, MediaType.***APPLICATION\_JSON***, mockHttpOutputMessage);

**return** mockHttpOutputMessage.getBodyAsString();

}

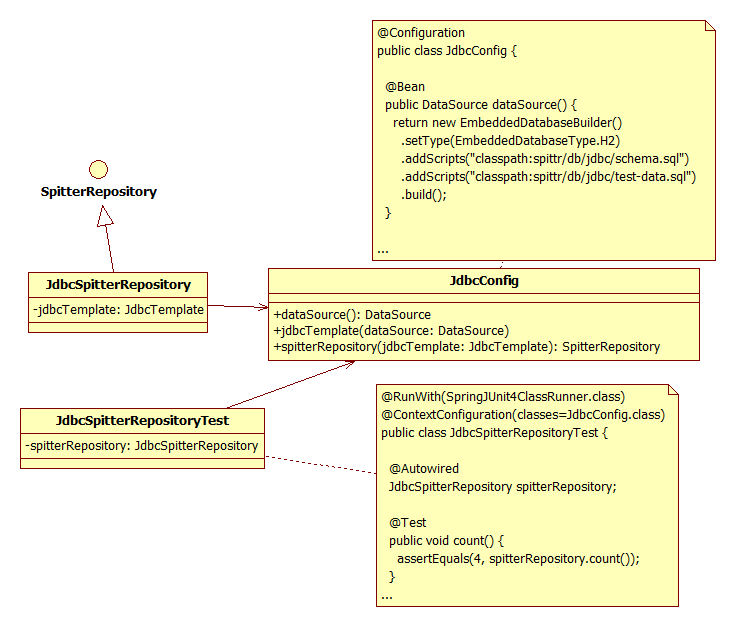
}

Reference:

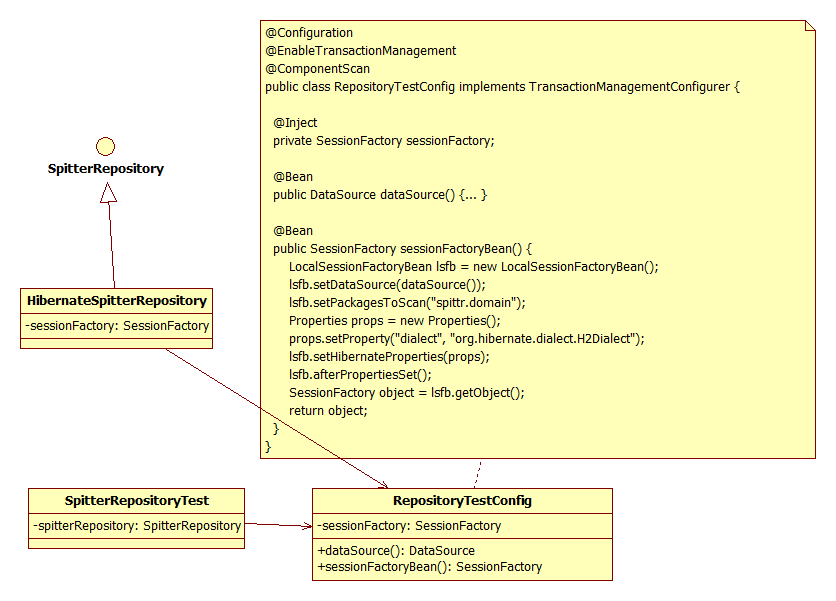
<http://spring.io/guides/tutorials/bookmarks/>

**Spring database**

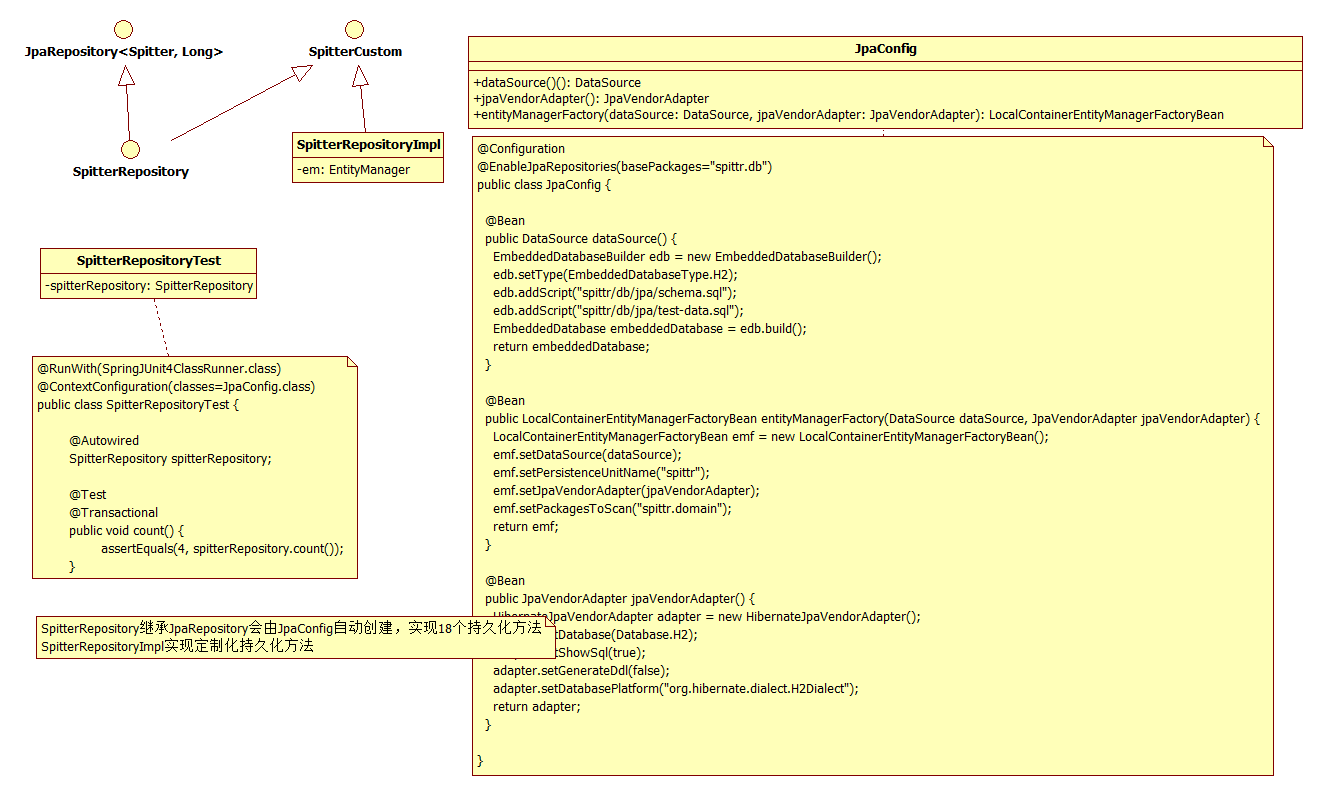
1. Jdbc



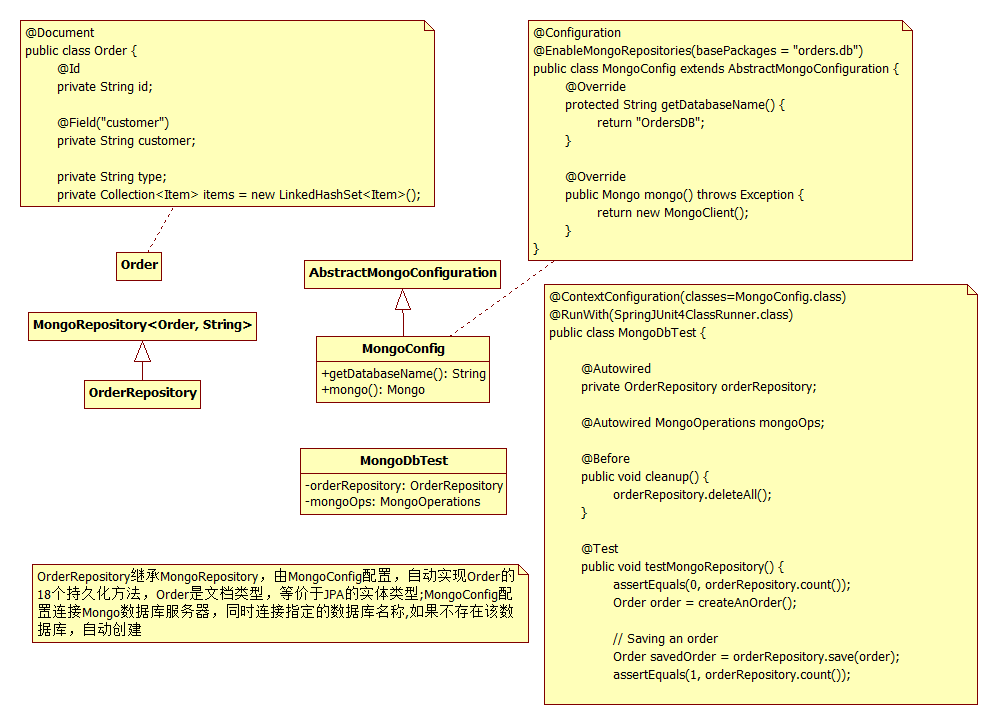
1. Hibernate



1. Spring data –JPA



1. **Spring data – MongoDB (highly importanc0e)**



MongoDb is an open-source NoSQL document database that uses a JSON-like schema instead of traditional table-based relational data.

Spring Data MongoDB brings MongoDB to Spring applications in three ways:

Annotations for object-to-document mapping

Template-based database access with MongoTemplate

Automatic runtime repository generation

**Spring – consume rest service**

1. Json 实体

@JsonIgnoreProperties(ignoreUnknown = **true**)

**public** **class** Page {

**private** String name;

**private** String about;

**private** String phone;

**private** String website;

**private** String mission ;

**private** String description ;

}

1. Access RESTful service via RestTemplate

**static** **final** String ***url*** = "http://graph.facebook.com/pivotalsoftware" ;

**public** **static** Page getPage() {

RestTemplate restTemplate = *getRestTemplate*();

Page page = restTemplate.getForObject(***url***, Page.**class**);

**return** page ;

}

**public** **static** RestTemplate getRestTemplate(){

SimpleClientHttpRequestFactory requestFactory = **new** SimpleClientHttpRequestFactory();

Proxy proxy= **new** Proxy(Proxy.Type.***HTTP***, **new** InetSocketAddress("161.92.51.225", 8080));

requestFactory.setProxy(proxy);

RestTemplate restTemplate = **new** RestTemplate(requestFactory);

**return** restTemplate ;

}

**Spring -- Scheduling Tasks**

1. component

@Component

**public** **class** ScheduledTasks {

**private** **static** **final** SimpleDateFormat ***dateFormat*** = **new** SimpleDateFormat("HH:mm:ss") ;

@Scheduled(fixedRate = 5000)

**public** **void** reportCurrentTime(){

System.***out***.println("This time is now " + ***dateFormat***.format(**new** Date())) ;

}

}

The Scheduled annotation defines when a particular method runs

1. configuration

@SpringBootApplication

@EnableScheduling

**public** **class** Application {

**public** **static** **void** main(String[] args) **throws** Exception{

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

}

}

[@EnableScheduling](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#scheduling-enable-annotation-support) ensures that a background task executor is created.

**Spring -- detect device**

1. controller

@RestController

**public** **class** DeviceDetectionController {

@RequestMapping("/detect-device")

**public** String detectDevice(Device device){

String deviceType = "unknown";

**if** (device.isNormal()) {

deviceType = "normal";

} **else** **if** (device.isMobile()) {

deviceType = "mobile";

} **else** **if** (device.isTablet()) {

deviceType = "tablet";

}

**return** "Hello " + deviceType + " browser!";

}

}

**spring – SSL**

JKS: Java密钥库(java keystore),JKS的Provider是SUN

JKS文件是一个java中的密钥管理库，里面可以放各种密钥文件.密钥库会有一把锁，是JKS文件的密码;里面存放的密钥也各有不同，每个密钥都有一个名字（叫别名），密钥分公钥和私钥，公钥只要你能进入仓库你就可以随便查看拿走，私钥则是有密码的，只允许有权限的人查看拿走。

%JAVA\_HOME%/bin/Keytool.exe 是Java数据证书的管理工具，数字证书是以一条一条(采用别名区别)的形式存入证书库的中，证书库中的一条证书包含该条证书的私钥，公钥和对应的数字证书的信息。证书库中的一条证书可以导出数字证书文件，数字证书文件只包括主体信息和对应的公钥。

// 创建密钥库及数字证书

cmd> keytool -genkey -alias tomcat -keyalg RSA -keystore d:\mykeystore.p12 -dname "CN=localhost, OU=localhost, O=localhost, L=SH, ST=SH, C=CN" -keypass changeit -storepass -validity 180

参数说明：

-genkey表示要创建一个新的密钥

-alias密钥的别名

-keyalg使用加密的算法，这里是RSA

-keystore 密钥保存在D:盘目录下的mykeystore.p12文件中

-dname表示密钥的Distinguished Names， 表明了密钥的发行者身份. CN=commonName; OU=organizationUnit; O=organizationName ; L=localityName ; S=stateName; C=country

-keypass私有密钥的密码,这里设置为changeit

-storepass 存取密码，这里设置为changeit，这个密码提供系统从mykeystore文件中将信息取出

-validity该密钥的有效期为 180天 (默认为90天)

//查看证书库中的所有数字证书 （需要密码）

cmd> keytool –list –rfc –keystore mykeystore.p12 –storetype pkcs12

// 查看证书详细 （需要密码）

cmd > keytool -list -v -keystore mykeystore.p12 -storetype pkcs12

//导出证书

cmd> keytool –export –alias tomcat –file mykeystore.cer –keystore mykeystore.p12 –storetype pkcs12

//导入证书至keystore文件中

cmd> keystore –import –v –alias tomcat –file keystore.pfx –keystore mykeystore.p12 –storetype pkcs12

//p12, pfx, bks, jks 互换

p12 -> pfx于文件采用相同的二进制格式，所以只要扩展名改一下就可以

p12 -> jks:

cmd> keytool -importkeystore -srckeystore [MY\_FILE.p12] -srcstoretype pkcs12

-srcalias [ALIAS\_SRC] -destkeystore [MY\_KEYSTORE.jks]

-deststoretype jks -deststorepass [PASSWORD\_JKS] -destalias [ALIAS\_DEST]

p12 -> bks

cmd> keytool –export –alias tomcat –keystore keystore.p12 –file keystore.crt

cmd> keytool –import –alias tomcat –file keystore.crt –keystore keystore.bks –storetype BKS –provider org.bouncycastle.jce.provider.BouncyCastleProvider

(注意：Java中配置加密组件Bouncy Castle, <http://wenku.baidu.com/view/4bdb903d0912a216147929c6.html>

)

PFX（PKCS#12）是公钥加密标准，它规定了可包含所有私钥、公钥和证书。其以二进制格式存储，在windows中可以直接导入到密钥区，注意，PKCS#12的密钥库保护密码同时也用于保护Key。

BKS来自BouncyCastleProvider，它使用的也是TripleDES来保护密钥库中的Key，它能够防止证书库被不小心修改（Keystore的keyentry改掉1个bit都会产生错误），BKS能够跟JKS互操作。

Enable HTTPS in spring boot:

1. Get yourself a SSL certificate: generate a self-signed certificate or get one from a Certificate Authority

cmd> keytool –genKey –alias tomcat –keyalg RSA –keystore keystore.p12 –dname “CN=localhost, OU=localhost, O=localhost, L=SH, ST=SH” –keypass changeit –storepass changeit –validity 180

1. Enable HTTPS in Spring Boot

//application.yml

server:

port: 9002

ssl:

key-store: classpath:keystore.p12

key-store-password: changeit

keyAlias: tomcat

key-password: changeit

1. Access HTTS

web explorer:

<https://localhost:9002>

postman: 先安装keystore.p12 (double click->…)

<https://localhost:9002> GET

Java client:

SSLConnectionSocketFactory socketFactory = **new** SSLConnectionSocketFactory(

**new** SSLContextBuilder() .loadTrustMaterial(**null**, **new** TrustSelfSignedStrategy()).build());

HttpClient httpClient = HttpClients.*custom*().setSSLSocketFactory(socketFactory)

.build();

TestRestTemplate testRestTemplate = **new** TestRestTemplate();

((HttpComponentsClientHttpRequestFactory) testRestTemplate.getRequestFactory()).setHttpClient(httpClient);

ResponseEntity<String> entity = testRestTemplate .getForEntity("https://localhost:" + **this**.port, String.**class**);

*assertEquals*(HttpStatus.***OK***, entity.getStatusCode());

Android client:

keystore.bks <- keystore.p12

RestAdapter restAdapter = new RestAdapter.Builder()  
 .setEndpoint(url)  
 .setClient(new OkClient(SSLOkHttpClient.*getSafeOkHttpClient*(false)))  
 .build() ;  
GreetingService service = restAdapter.create(GreetingService.class) ;  
Greeting greeting = service.greeting("QiZhong Lin") ;  
return greeting ;

public class SSLOkHttpClient {  
  
 private static final String *SERVER\_IP* = "161.92.141.144";  
 private static final int *SERVER\_PORT* = 9002;  
 private static final String *CLIENT\_TRUST\_MANAGER* = "X509";  
 private static final String *CLIENT\_TRUST\_PASSWORD* = "changeit";  
 private static final String *CLIENT\_AGREEMENT* = "TLS";  
 private static final String *CLIENT\_TRUST\_KEYSTORE* = "BKS";  
 private static final String *CLIENT\_CACERT\_BKS* = "res/raw/keystore.bks";  
  
 public static OkHttpClient getSafeOkHttpClient(final boolean verifyHostname) {  
 try {  
 TrustManagerFactory trustManager = TrustManagerFactory.*getInstance*(*CLIENT\_TRUST\_MANAGER*);  
 KeyStore tks = KeyStore.*getInstance*(*CLIENT\_TRUST\_KEYSTORE*);  
 InputStream is = SSLOkHttpClient.class.getClassLoader().getResourceAsStream(*CLIENT\_CACERT\_BKS*);  
 tks.load(is, *CLIENT\_TRUST\_PASSWORD*.toCharArray());  
 trustManager.init(tks);  
  
 final SSLContext sslContext = SSLContext.*getInstance*(*CLIENT\_AGREEMENT*);  
 sslContext.init(null, trustManager.getTrustManagers(), new java.security.SecureRandom());  
 final SSLSocketFactory sslSocketFactory = sslContext.getSocketFactory();  
  
 OkHttpClient okHttpClient = new OkHttpClient();  
 okHttpClient.setSslSocketFactory(sslSocketFactory);  
 okHttpClient.setHostnameVerifier(new HostnameVerifier() {  
 @Override  
 public boolean verify(String hostname, SSLSession session) {  
 if(verifyHostname){  
 X509Certificate[] peerCerts;  
 try {  
 peerCerts = session.getPeerCertificateChain();  
 } catch (SSLPeerUnverifiedException e) {  
 return false;  
 }  
 for (X509Certificate peerCert : peerCerts) {  
 Principal x500s = peerCert.getSubjectDN();  
 try {  
 peerCert.checkValidity();  
 } catch (Exception e) {  
 return false;  
 }  
 }  
 return true;  
 } else {  
 return true;  
 }  
 }  
 });  
 return okHttpClient;  
 } catch (Exception e) {  
 throw new RuntimeException(e);  
 }  
 }  
}

**spring – 外部化配置**

管理外部化的配置并将其转换为对象结构，这个对象可以在整个应用上下文中使用。创建一个简单老式的Java/Groovy对象（Plain Old Java/Groovy Object），并使用@ConfigurationProperties注解，那么这个对象就能使用Boot配置结构中预先定义的name名下的配置项

//application.yml (配置文件)

application:

name: sb-ms-custdepl

version: 0.1-CUSTOMER

metrics:

dbExecutionTimeKey: user.get.db.time

// ApplicationProperties.java (映射对象)

@ConfigurationProperties(name = "application")

class ApplicationProperties {

String name

String version

final Metrics metrics = new Metrics()

static class Metrics {

String dbExecutionTimeKey

}

}

// AppInfo.Controller.java (依赖注入)

@RestController

@Configuration

@RequestMapping("/appinfo")

@EnableAutoConfiguration

class AppInfoController {

@Autowired

ApplicationProperties applicationProperties

@Value('${application.metrics.dbExecutionTimeKey}')

String dbExecutionKey

@RequestMapping(method=[RequestMethod.GET])

def get() {

[

name: applicationProperties.name,

version: applicationProperties.version

]

}

@Bean

ApplicationProperties applicationProperties() {

new ApplicationProperties()

}

public static void main(String[] args) {

SpringApplication.run UserController, args

}

}

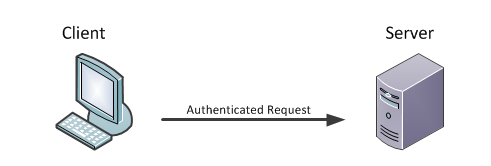
**spring – Authentication**

in order for these applications to access user data on other sites, they ask for usernames and passwords. Not only does this require exposing user passwords to someone else – often the same passwords used for online banking and other sites – it also provides these application unlimited access to do as they wish. They can do anything, including changing the passwords and lock users out.

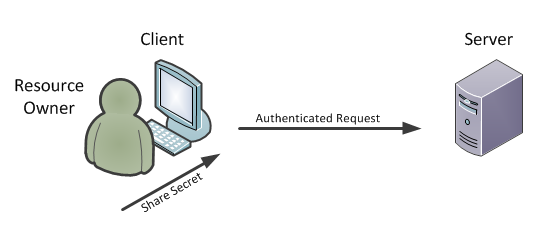
OAuth provides a method for users to grant third-party access to their resources without sharing their passwords. It also provides a way to grant limited access (in scope, duration, etc.).

In order for the client to access resources, it first has to obtain permission from the resource owner. This permission is expressed in the form of a token and matching shared-secret. The purpose of the token is to make it unnecessary for the resource owner to share its credentials with the client. Unlike the resource owner credentials, tokens can be issued with a restricted scope and limited lifetime, and revoked independently.

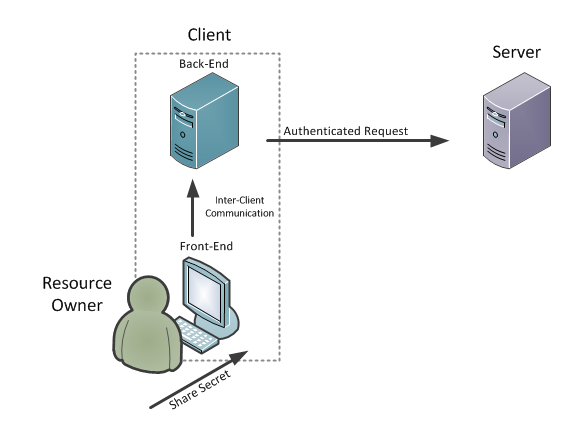
OAuth defines three roles: client, server, and resource owner,These three roles are present in any OAuth transaction; in some cases the client is also the resource owner. The original version of the specification used a different set of terms for these roles: consumer (client), service provider (server), and user (resource owner).



In the traditional client-server authentication model, the client uses its credentials to access its resources hosted on the server. As far as the server is concerned, the shared secret used by the client belongs to the client, The server doesn’t really care where it came from or if the client is acting on behalf of some other entity. As long as the shared secret matches the server’s expectation, the request is processed.



There are many times when the client is acting on behalf of another entity. That entity can be another machine or person.it is enough that entity requires resource owner's credentials to make requests – pretending to be the resource owner. because User credentials typically include a username or screen-name and a password, which contains too much power, in tech term, resource owner's credentials has different role compared to user's credentials



Protected Resources

A protected resource is a resource stored on (or provided by) the server which requires authentication in order to access it. Protected resources are owned or controlled by the resource owner. Anyone requesting access to a protected resource must be authorized to do so by the resource owner (enforced by the server).

A protected resource can be data (photos, documents, contacts), services (posting blog item, transferring funds), or any resource requiring access restrictions. While OAuth can be used with other transport protocols, it is only defined for HTTP(S) resources.

Credentials and Tokens

OAuth uses three kinds of credentials: client credentials, temporary credentials, and token credentials. The original version of the specification used a different set of terms for these credentials: consumer key and secret (client credentials), request token and secret (temporary credentials), and access token and secret (token credentials). The specification still uses a parameter name ‘oauth\_consumer\_key‘ for backwards compatibility.

The client credentials are used to authenticate the client. This allows the server to collect information about the clients using its services, offer some clients special treatment such as throttling-free access, or provide the resource owner with more information about the clients seeking to access its protected resources. In some cases, the client credentials cannot be trusted and can only be used for informational purposes only, such as in desktop application clients.

Token credentials are used in place of the resource owner’s username and password. Instead of having the resource owner share its credentials with the client, it authorizes the server to issue a special class of credentials to the client which represent the access grant given to the client by the resource owner. The client uses the token credentials to access the protected resource without having to know the resource owner’s password.

Token credentials include a token identifier, usually (but not always) a random string of letters and numbers that is unique, hard to guess, and paired with a secret to protect the token from being used by unauthorized parties. Token credentials are usually limited in scope and duration, and can be revoked at any time by the resource owner without affecting other token credentials issued to other clients.

The OAuth authorization process also uses a set of temporary credentials which are used to identify the authorization request. In order to accommodate different kind of clients (web-based, desktop, mobile, etc.), the temporary credentials offer additional flexibility and security.

In OAuth 1.0, the secret half of each set of credentials is defined as a symmetric shared secret. This means that both the client and server must have access to the same secret string. However, OAuth supports an RSA-based authentication method which uses an asymmetric client secret

HTTP defines an authentication scheme called ‘Basic’ which is commonly used by many sites and APIs(Spring Boot automatically secures all HTTP endpoints with "basic" authentication). The way ‘Basic’ works is by sending the username and password in plain text with each request. When not used over HTTPS, ‘Basic’ suffers from significant security risks. First, it transmits passwords unencrypted which allows anyone listening to capture and reuse those credentials.

The OAuth signature method was primarily designed for insecure communications — mainly non-HTTPS. HTTPS is the recommended solution to prevent a man-in-the-middle attack (MITM), eavesdropping, and other security risks. However, HTTPS is often not available. When OAuth is used over HTTPS, it offers a simple method for a more efficient implementation called PLAINTEXT which offloads most of the security requirements to the HTTPS layer. It is important to understand that PLAINTEXT should not be used over an insecure channel

The signature and shared secret provide some level of security but are still vulnerable to attacks. What is missing is something to prevent requests intercepted by an unauthorized party, usually by sniffing the network, from being reused. This is known as a replay attack.They will however, be able to make the same sign request over and over again. If the intercepted request provides access to sensitive protected data, it can be a significant security risk.

To prevent compromised requests from being used again (replayed), OAuth uses a nonce and timestamp.

Resource Server (like api-gateway)

To use the access token you need a Resource Server (which can be the same as the Authorization Server). Creating a Resource Server is easy, just add @EnableResourceServer and provide some configuration to allow the server to decode access tokens. If your application is also an Authorization Server it already knows how to decode tokens, so there is nothing else to do

com.philips.pscs.cpc.gateway.security. CustomWebSecurityConfigurerAdapter.java

@Configuration

**public** **class** CustomWebSecurityConfigurerAdapter **extends**

WebSecurityConfigurerAdapter {

@Autowired

**private** SecurityConfig securityConfig;

@Override

**protected** **void** configure(HttpSecurity http) **throws** Exception {

http.httpBasic().disable();

http.csrf().disable();

}

@Bean

**public** ResourceServerTokenServices tokenService() {

RemoteTokenServices tokenServices = **new** RemoteTokenServices();

tokenServices.setClientId(securityConfig.getClientId());

tokenServices.setClientSecret(securityConfig.getClientSecret());

tokenServices.setCheckTokenEndpointUrl(securityConfig.getAuthServer()

+ "oauth/check\_token");

**return** tokenServices;

}

@Override

**public** AuthenticationManager authenticationManagerBean() **throws** Exception {

OAuth2AuthenticationManager authenticationManager = **new** OAuth2AuthenticationManager();

authenticationManager.setTokenServices(tokenService());

**return** authenticationManager;

}

@Configuration

@EnableResourceServer

**protected** **static** **class** ResourceServerConfiguration **extends**

ResourceServerConfigurerAdapter {

@Autowired

**private** SecurityConfig securityConfig;

@Override

**public** **void** configure(ResourceServerSecurityConfigurer resources) {

resources.resourceId(securityConfig.getResourceId());

}

@Override

**public** **void** configure(HttpSecurity http) **throws** Exception {

http.authorizeRequests().antMatchers("/v1/ccs/\*\*").authenticated();

}

}

}

application.yml

authSecurity:

authServer: http://localhost:9005/

clientId: clientapp

clientSecret: 123456

resourceId: restservice

Authorization Server (like authorization-service)

To create an Authorization Server and grant access tokens you need to use @EnableAuthorizationServer and provide security.oauth2.client.client-id and security.oauth2.client.client-secret] properties. The client will be registered for you in an in-memory repository.

Having done that you will be able to use the client credentials to create an access token, for example:

cmd> curl client:secret@localhost:8080/oauth/token -d grant\_type=password -d username=user -d password=pwd

The basic auth credentials for the /token endpoint are the client-id and client-secret. The user credentials are the normal Spring Security user details (which default in Spring Boot to “user” and a random password).

com.philips.pscs.cpc.authorization.security.CustomAuthenticationServer

@Configuration

@EnableAuthorizationServer

**public** **class** CustomAuthenticationServer **extends** AuthorizationServerConfigurerAdapter {

**private** TokenStore tokenStore = **new** InMemoryTokenStore();

@Autowired

**private** SecurityConfig securityConfig;

@Autowired

@Qualifier("authenticationManagerBean")

**private** AuthenticationManager authenticationManager;

//配置用户存储信息 (数据库)

@Autowired

**private** CustomUserDetailsService userDetailsService;

@Override

**public** **void** configure(AuthorizationServerEndpointsConfigurer endpoints)

**throws** Exception {

endpoints

.tokenStore(**this**.tokenStore)

.authenticationManager(**this**.authenticationManager)

.userDetailsService(userDetailsService)

.accessTokenConverter(accessTokenConverter())

.tokenEnhancer(tokenEnhancer());

}

//配置返回的token对象（除默认信息，还可添加自定义信息）

@Bean

**public** TokenEnhancer tokenEnhancer() {

**return** **new** CustomTokenEnhancer();

}

@Bean

**public** DefaultAccessTokenConverter accessTokenConverter() {

**return** **new** DefaultAccessTokenConverter();

}

@Override

**public** **void** configure(ClientDetailsServiceConfigurer clients) **throws** Exception {

clients

.inMemory()

.withClient(securityConfig.getClientId()) .authorizedGrantTypes(SecurityConstants.***GrantType\_Password***, SecurityConstants.***GrantType\_RefreshToken***) .scopes(SecurityConstants.***Scope\_Read***, SecurityConstants.***Scope\_Write***)

.resourceIds(securityConfig.getResourceId())

.secret(securityConfig.getClientSecret());

}

@Override

**public** **void** configure(AuthorizationServerSecurityConfigurer oauthServer) **throws** Exception {

oauthServer.checkTokenAccess("permitAll()");

}

@Bean

@Primary

**public** DefaultTokenServices tokenServices() {

DefaultTokenServices tokenServices = **new** DefaultTokenServices();

tokenServices.setTokenEnhancer(tokenEnhancer());

tokenServices.setSupportRefreshToken(**true**);

tokenServices.setTokenStore(**this**.tokenStore);

**return** tokenServices;

}

}

CustomUserDetailsService.java

@Service

**public** **class** CustomUserDetailsService **implements** UserDetailsService {

//private final UserRepository userRepository;

**private** UserDao userRepository;

@Autowired

**public** CustomUserDetailsService(UserDao userRepository) {

**this**.userRepository = userRepository;

}

@Override

**public** UserDetails loadUserByUsername(String username) **throws** UsernameNotFoundException {

User user = userRepository.getUser(username);

**if** (user == **null**) {

**throw** **new** UsernameNotFoundException(String.*format*("User %s does not exist!", username));

}

user.setPassword(decryptPassword(user.getPassword()));

**return** **new** UserRepositoryUserDetails(user);

}

**private** String decryptPassword(String password) {

**byte**[] passwordBytes = password.getBytes();

**byte**[] decodedBytes = Base64.*decodeBase64*(passwordBytes);

String decodedPassword = **new** String(decodedBytes);

**return** decodedPassword;

}

**private** **final** **static** **class** UserRepositoryUserDetails **extends** User **implements** UserDetails {

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

**private** UserRepositoryUserDetails(User user) {

**super**(user);

}

@Override

**public** Collection<? **extends** GrantedAuthority> getAuthorities() {

**return** getRoles();

}

@Override

**public** String getUsername() {

**return** getLoginId();

}

@Override

**public** **boolean** isAccountNonExpired() {

**return** **true**;

}

@Override

**public** **boolean** isAccountNonLocked() {

**return** **true**;

}

@Override

**public** **boolean** isCredentialsNonExpired() {

**return** **true**;

}

@Override

**public** **boolean** isEnabled() {

**return** **true**;

}

}

}CustomTokenEnhancer.java

@Configuration

**public** **class** CustomTokenEnhancer **implements** TokenEnhancer {

@Override

**public** OAuth2AccessToken enhance(OAuth2AccessToken accessToken, OAuth2Authentication authentication) {

DefaultOAuth2AccessToken result = **new** DefaultOAuth2AccessToken(accessToken);

User user = (User) authentication.getPrincipal();

**final** Map<String, Object> additionalInformation = **new** HashMap<>();

additionalInformation.put("firstName", user.getFirstName());

additionalInformation.put("lastName", user.getLastName());

additionalInformation.put("loginId", user.getLoginId());

additionalInformation.put("roles", AuthorityUtils.*authorityListToSet*(authentication.getAuthorities()));

result.setAdditionalInformation(additionalInformation);

**return** result;

}

}

application.yml

authSecurity:

clientId: clientapp

clientSecret: 123456

resourceId: restservice

**Spring data rest**

Spring Data REST builds on top of **Spring Data repositories**, analyzes your application's **domain model** and **exposes hypermedia-driven HTTP resources** for aggregates contained in the model.

Via hypermedia-driven interface, there is no need to exchange a formal contract or interface document with your customers. A hypermedia-driven site provides information to navigate the site's REST interfaces dynamically by including hypermedia links with the responses. This capability differs from that of SOA-based systems and WSDL-driven interfaces. With SOA, servers and clients usually must access a fixed specification that might be staged somewhere else on the website, on another website, or perhaps distributed by email.

1. >mongod –dbpath ./data //启动mongo db
2. build.gradle

compile("org.springframework.boot:spring-boot-starter-data-rest")

compile("org.springframework.boot:spring-boot-starter-data-mongodb")

compile 'org.springframework.data:spring-data-rest-hal-browser'

1. create a domain object

src/main/java/com.philips.rest.domain.Person.java

public class Person {

@Id private String id ;

private String firstName ;

private String lastName ;

private String age ;

}

1. create repository

src/main/java/com.philips.rest.db.PersonRepository.java

public interface PersonRepository extends MongoRepository<Person, String>{

List<Person> findByLastName(@Param("name") String name) ;

}

1. run

e.1 expose RESTful API via HAL(hypertext application language) format

|  |  |  |
| --- | --- | --- |
| Path | Method | Header |
| /persons{?page,size,sort}  (等于默认：?page=0&size=20) | GET |  |
| /persons/  {  “firstName”: “QiZhong”,  “lastName”: “Lin”,  “age”: 36  } | POST | Content-Type: application/json |
| /persons/{id} | GET |  |
| /persons/{id}  {  “firstName”: “QiZhong Jr.”,  “lastName”: “Lin”,  “age”: 3  } | PUT | Content-Type: application/json |
| /persons/{id}  {  “firstName”: “QiZhong Jr.”  } | PATCH | Content-Type: application/json |
| /persons/{id} | DELETE |  |
| /persons/search/findByLastName?name={name} | GET |  |

e.2 resource link (from root URL to deep), tools: HAL Browser

localhost:8080/

1. Spring HATEOAS

HATEOAS (Hypermedia as the Engine of Application State) is a constraint of the REST application architecture.

A hypermedia-driven site provides information to navigate the site's REST interfaces dynamically by including hypermedia links with the responses. This capability differs from that of SOA-based systems and WSDL-driven interfaces. With SOA, servers and clients usually must access a fixed specification that might be staged somewhere else on the website, on another website, or perhaps distributed by email.

With HATEOAS, the output makes it easy to glean how to interact with the service without looking up a specification or other external document.

According to the Richardson Maturity Model, HATEOAS is considered the final level of REST. This means that each link is presumed to implement the standard REST verbs of GET, POST, PUT, and DELETE (or a subset). Thus providing the links as shown above gives the client the information they need to navigate the service.

1. build.gradle

compile("org.springframework.boot:spring-boot-starter-hateoas")

1. create domain object

public class Greeting extends ResourceSupport {

private final String content;

@JsonCreator

public Greeting(@JsonProperty("content") String content) {

this.content = content;

}

public String getContent() {

return content;

}

}

1. Create a resource controller

@Controller

public class GreetingController {

private static final String TEMPLATE = "Hello, %s!";

@RequestMapping("/greeting")

@ResponseBody

public HttpEntity<Greeting> greeting(

@RequestParam(value = "name", required = false, defaultValue = "World") String name) {

Greeting greeting = new Greeting(String.format(TEMPLATE, name));

greeting.add(linkTo(methodOn(GreetingController.class).greeting(name)).withSelfRel());

return new ResponseEntity<Greeting>(greeting, HttpStatus.OK);

}

}

**Spring Data – JPA (relational database)**

1. build.gradle

compile("org.springframework.boot:spring-boot-starter-data-jpa")

1. create a domain object

src/main/java/com.philips.jpa.domain.Customer.java

**@Entity**

public class Customer {

**@Id**

**@GeneratedValue(strategy=GenerationType.AUTO)**

private long id ;

private String firstname ;

private String lastname ;

protected Customer() {}

public Customer(String firstname, String lastname){

this.firstname = firstname ;

this.lastname = lastname ;

}

}

1. create repository

src/main/java/com.philips.jpa.db.CustomerRepository.java

public interface CustomerRepository extends CrudRepository<Customer, Long>{

public List<Customer> findByFirstname(String firstname) ;

public List<Customer> findByLastname(String lastname) ;

}

1. run

src/main/java/com.philips.jpa.Application.java

@SpringBootApplication

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

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

@Bean

public CommandLineRunner demo(CustomerRepository repository){

return (args) -> {

//save a couple of customers

repository.save(new Customer("Jack", "Bauer")) ;

repository.save(new Customer("Chloe", "O' Brain")) ;

// fetch all customers

repository.findAll().forEach((customer) -> log.info(customer.toString()));

// fetch an individual customer by ID

log.info(repository.findOne(1L).toString()) ;

// fetch customers by last name

repository.findByLastname("Bauer").forEach((item) -> log.info(item.toString()));

} ;

}

}

By default, Spring Boot will enable JPA repository support and look in the package (and its subpackages) where @SpringBootApplication is located

**in-memory database:** Spring Boot can auto-configure embedded H2, HSQL and Derby databases. You don’t need to provide any connection URLs, simply include a build dependency to the embedded database that you want to use. By default, JPA databases will be automatically created only if you use an embedded database (H2, HSQL or Derby).

Using H2’s web console (refer to Spring boot reference guide): /h2-console

**production database:** DataSource configuration is controlled by external configuration properties in spring.datasource.\*

application.yml:

spring:

datasource:

driver-class-name: org.postgresql.ds.PGSimpleDataSource

url: jdbc:postgresql://localhost:5432/bookstore

username: postgres

password: qzlin

You often won’t need to specify the driver-class-name since Spring boot can deduce it for most databases from the url.