Introduction to Spring Framework

August 2014

Serhat CAN
can.srht@gmail.com

Content

What is Spring Framework?

- Spring is **the most popular application development framework** for **enterprise** Java.
- Open source Java platform since 2003.
- Spring supports all major application servers and JEE standards.
- Spring handles the infrastructure so you can focus on your application.

Dependency Injection

Introduction to Concept

- The technology that actually defines Spring (Heart of Spring).
- Dependency Injection helps us to keep our classes as indepedent as possible.
    - Increase reuse by applying low coupling
    - Easy testing
    - More understandable

Dependency Injection
Introduction to Concept

An injection is the passing of a dependency (a service) to a dependent object (a client). Passing the service to the client, rather than allowing a client to build or find the service, is the fundamental requirement of the pattern.

“Dependency injection is a pattern where the container passes objects by name to other objects, via either constructors, properties, or factory methods.”

Dependency Injection
Relationship Between DI and Inversion of Control

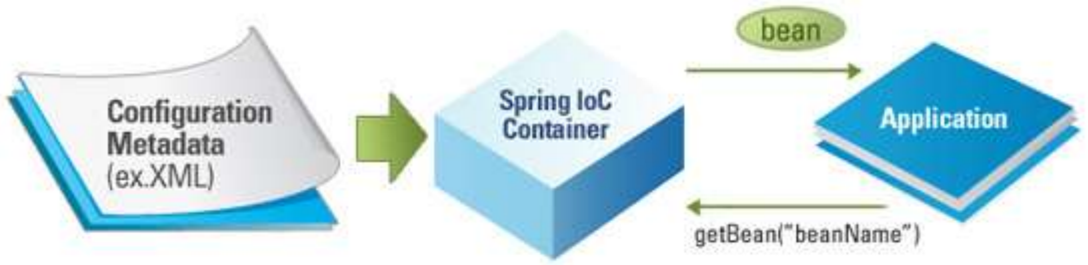
In software engineering, inversion of control (IoC) describes a design in which custom-written portions of a computer program receive the flow of control from a generic, reusable library.

The Inversion of Control (IoC) is a general concept, and it can be expressed in many different ways and dependency Injection is merely one concrete example of Inversion of Control.

Dependency Injection

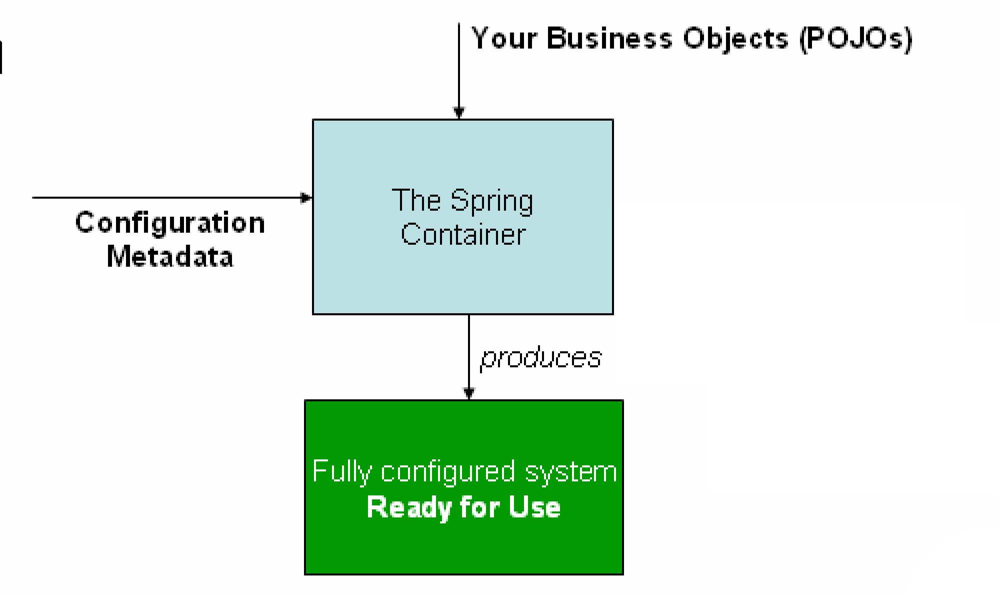
IoC Container

- The Spring container (IoC Container) is at the core of the Spring Framework.
- The container will create the objects, wire them together, configure them, and manage their complete lifecycle from creation till destruction.



Dependency Injection
IoC Container

- The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata provided.
- The configuration metadata can be represented either by;
  - XML,
  - Java annotations,
  - Java code.



Dependency Injection

Code Example

```java
package basicinjection;

public class Foo {
private String name;

public Foo() {   }

public Foo(String name) {
  this.name = name;
}

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

public String getName() {
  return name;
}


}
```

```java
package basicinjection;

public class Bar {
private String name;
private int age;
private Foo foo;

public Bar() {}

public Bar(String name,int age) {
  this.name = name;
  this.age = age;
}
public void setFoo(Foo foo) {
  this.foo = foo;
}

public void processFooName(){
  System.out.println("Name in Injected Foo is: "+foo.getName());
}

@Override
public String toString() {
  return "Bar has name = "+this.name+" and age = "+this.age;
}
}
```

To instantiate the above classes, one way is to do the usual new operator like new Foo() or new Bar() OR we can use the Spring dependency injection to instantiate these classes and set the properties accordingly.

Dependency Injection

Code Example

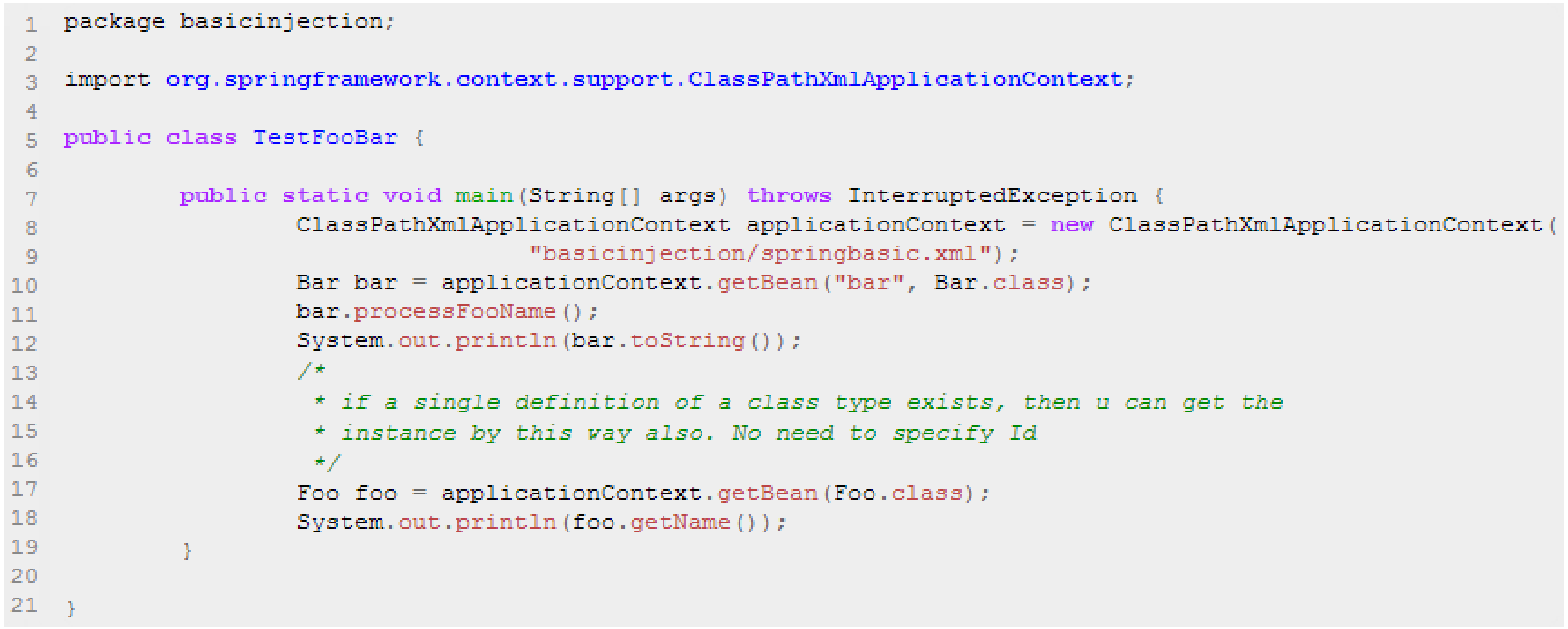
```xml
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

   <bean id="foo" class="basicinjection.Foo" scope="prototype">
          <constructor-arg index="0" value="Cleopatra"></constructor-arg>
   </bean>
   <bean id="bar" class="basicinjection.Bar">
          <constructor-arg type="int" value="26" />
          <constructor-arg type="java.lang.String" value="Arthur" />
          <property name="foo" ref="foo"></property>
   </bean>
</beans>
```

```java
Foo f = new Foo("Cleopatra");
```

```java
Bar b = new Bar("Arthur",26);
b.setFoo(f);
```

Dependency Injection

Code Example

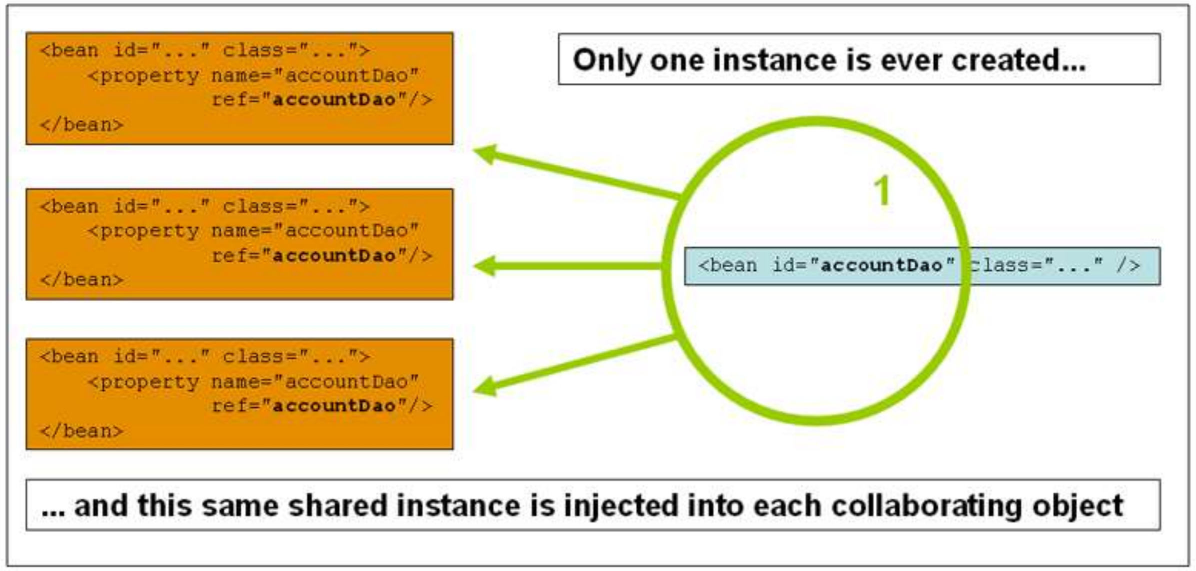


```java
package basicinjection;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestFooBar {

        public static void main(String[] args) throws InterruptedException {
                ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplicationContext(
                                "basicinjection/springbasic.xml");
                Bar bar = applicationContext.getBean("bar", Bar.class);
                bar.processFooName();
                System.out.println(bar.toString());
                /*
                 * if a single definition of a class type exists, then u can get the
                 * instance by this way also. No need to specify Id
                 */
                Foo foo = applicationContext.getBean(Foo.class);
                System.out.println(foo.getName());
        }

}
```

Spring's ClassPathXmlApplicationContext is the commonly used object that hold the information of all the beans that it instantiates.

Dependency Injection
Bean Scopes

Scope	Description
Singleton	(Default) Scopes a single bean definition to a single object instance per Spring IoC container.
Prototype	Scopes a single bean definition to any number of object instances.



Only one instance is ever created...
1
... and this same shared instance is injected into each collaborating object

```
<bean id="..." class="...">
    <property name="accountDao"
              ref="accountDao"/>
</bean>
```

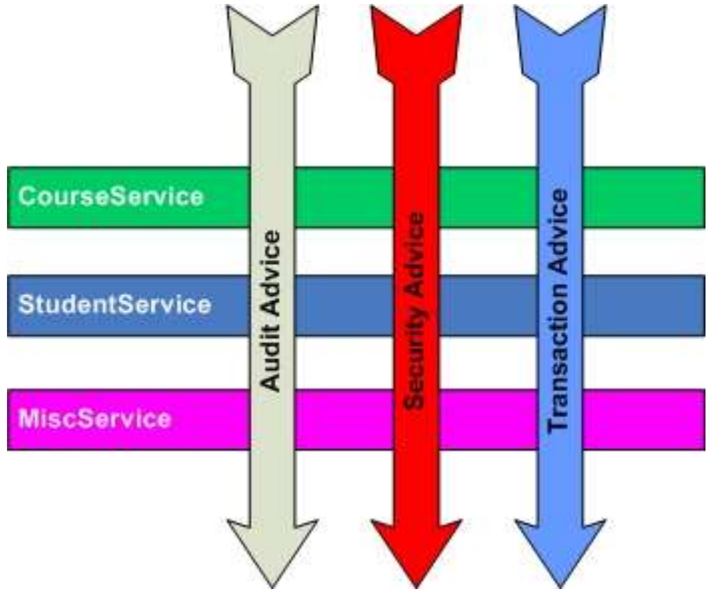
```
<bean id="accountDao" class="..." />
```

Aspect Oriented Programming (AOP)

Introduction to Concept

- AOP entails breaking down program logic into distinct parts called **concerns**.
- The functions that span multiple points of an application are called **cross-cutting concerns** and these cross-cutting concerns are conceptually separate from the application's business logic.
- AOP is like triggers in programming languages

  such as Perl, .NET, Java and others.



Examples of cross-cutting concerns:

- Logging
- Security
- Transaction
- Caching

Aspect Oriented Programming (AOP)

Spring AOP

- Spring AOP module provides interceptors to intercept an application, for example, when a method is executed, you can add extra functionality before or after the method execution.

- Spring AOP's approach to AOP differs from that of most other AOP frameworks. The aim is to provide a close integration between AOP implementation and Spring IoC, not to provide the most complete AOP implementation.

- Spring Framework's AOP functionality is normally used in conjunction with the Spring IoC container. Aspects are configured using normal bean definition syntax.

Aspect Oriented Programming (AOP)

Code Example

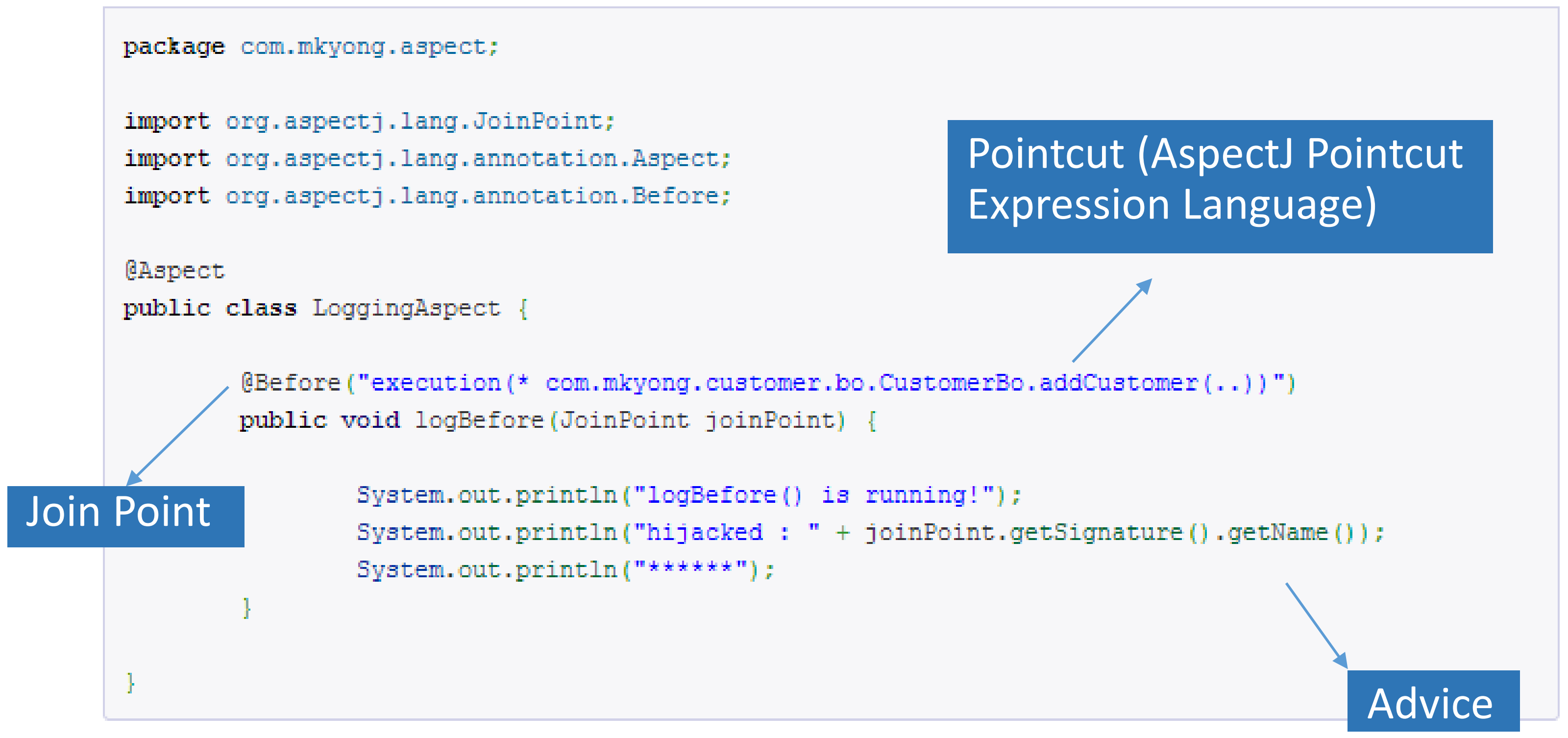
File : Spring-Customer.xml

```xml
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:aop="http://www.springframework.org/schema/aop"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-3.0.xsd ">

        <aop:aspectj-autoproxy />

        <bean id="customerBo" class="com.mkyong.customer.bo.impl.CustomerBoImpl" />

        <!-- Aspect -->
        <bean id="logAspect" class="com.mkyong.aspect.LoggingAspect" />

</beans>
```

Aspect Oriented Programming (AOP)

Code Example

File : LoggingAspect.java



```
package com.mkyong.aspect;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class LoggingAspect {

        @Before("execution(* com.mkyong.customer.bo.CustomerBo.addCustomer(..))")
        public void logBefore(JoinPoint joinPoint) {

                System.out.println("logBefore() is running!");
                System.out.println("hijacked : " + joinPoint.getSignature().getName());
                System.out.println("******");
        }

}
```

- **@Before** – Run before the method execution
- **@After** – Run after the method returned a result
- **@AfterReturning** – Run after the method returned a result, intercept the returned result as well.
- **@Around** – Run around the method execution, combine all three advices above.
- **@AfterThrowing** – Run after the method throws an exception

Aspect Oriented Programming (AOP)
Code Example

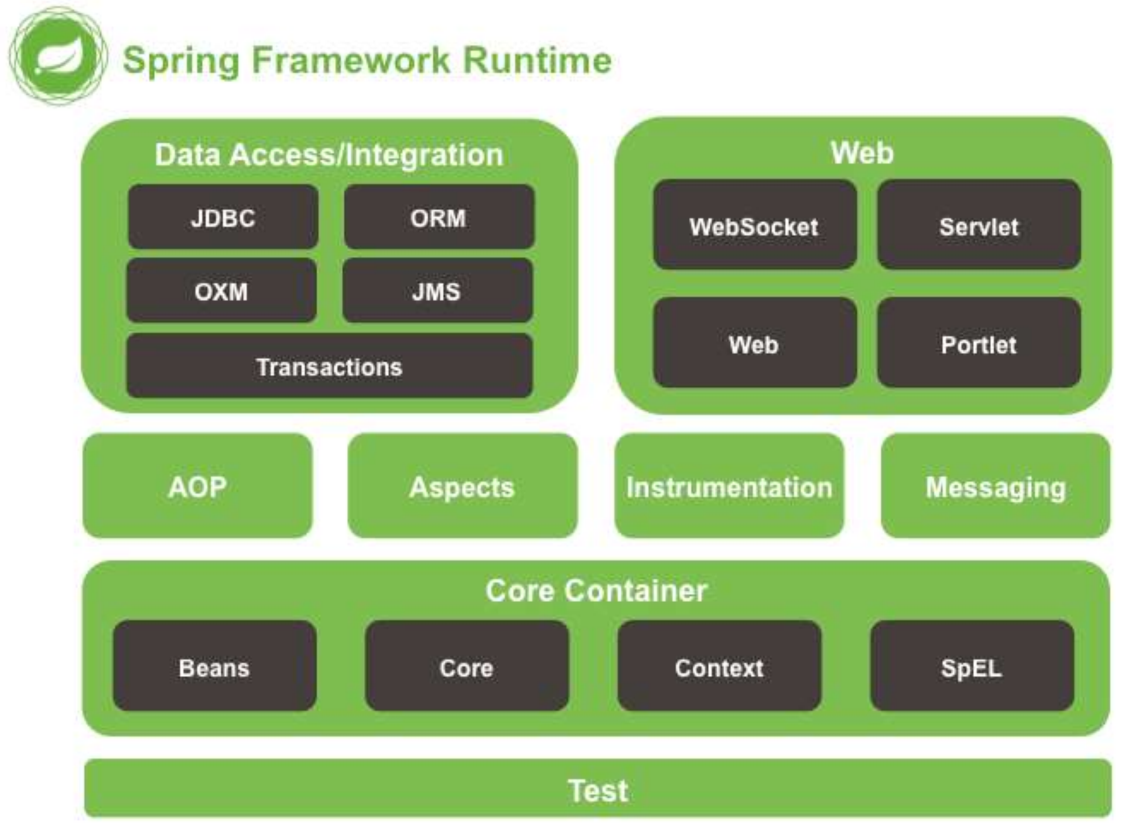
Run it

```
        CustomerBo customer = (CustomerBo) appContext.getBean("customerBo");
        customer.addCustomer();
```

Output

```
logBefore() is running!
hijacked : addCustomer
******
addCustomer() is running
```

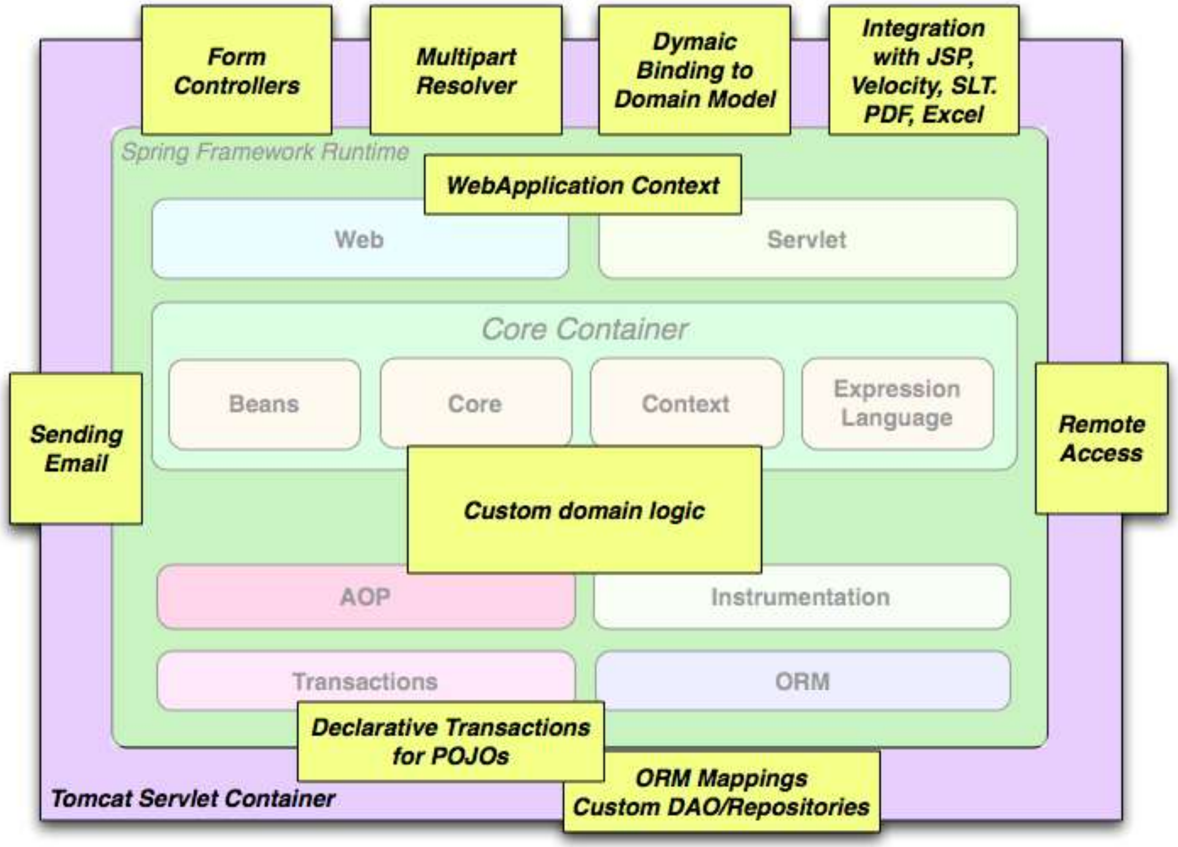
Spring Modules



Overview of the Spring Framework

- The Spring Framework consists of features organized into about 20 modules.
- These modules are grouped into Core Container, Data Access/Integration, Web, AOP (Aspect Oriented Programming), Instrumentation, and Test.

Spring Modules



Typical full-fledged Spring web application

- The building blocks described previously make Spring a logical choice in many scenarios, from applets to full-fledged enterprise applications that use Spring's transaction management functionality and web framework integration.

Spring Modules

Spring Projects

- Spring XD
- Spring Data
- Spring Integration
- Spring Batch
- Spring Security
- Spring Cloud
- Spring AMQP
- Spring Grails
- Spring Mobile
- Spring Social
- Spring for Android
- Spring Web Flow
- Spring LDAP
- Spring Groovy
- Spring Hateoas
- Spring Security OAuth

Advantages of Using Spring Framework

- Open source
- Lightweight and fast
- Moduler structure
- Low coupling thanks to Dependency Injection
- Resuable software
- AOP support
- Stable and lots of resources
- Projects that make our life easier like Spring Security

Introduction to Spring Framework - Part 2

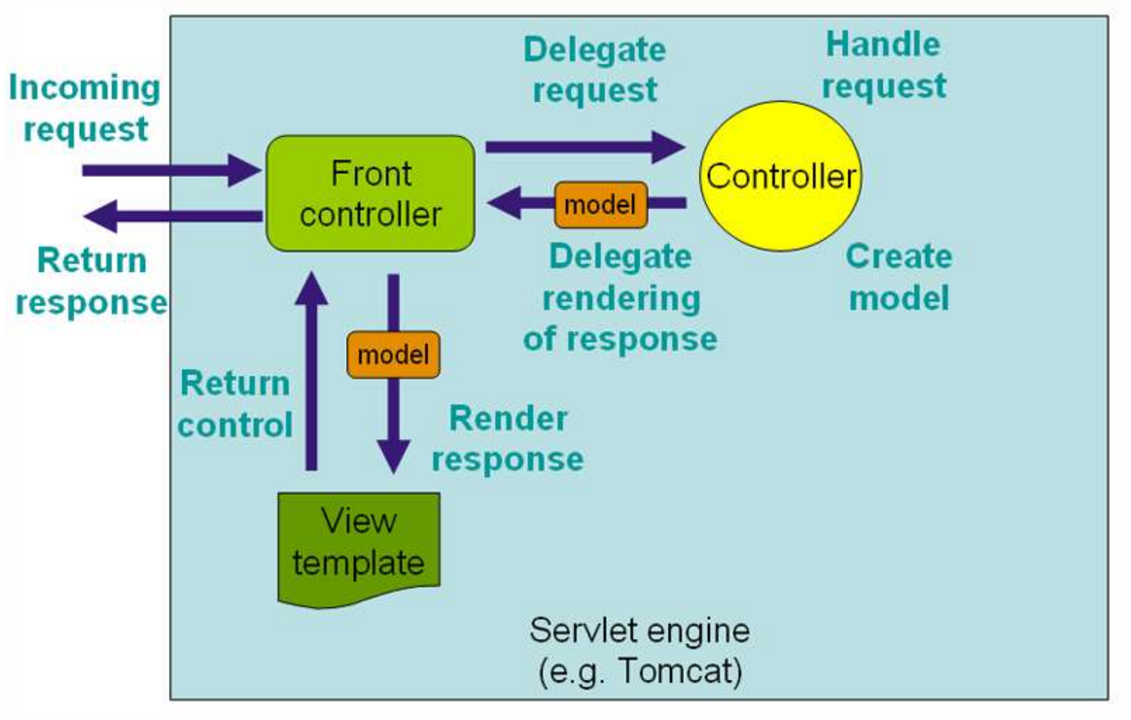
Content

- Spring MVC
- Spring RESTful Services
- Spring Security
- Spring Test

Spring MVC

- The Spring web MVC framework provides model-view-controller architecture and ready components that can be used to develop flexible and loosely coupled web applications.

- The MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.



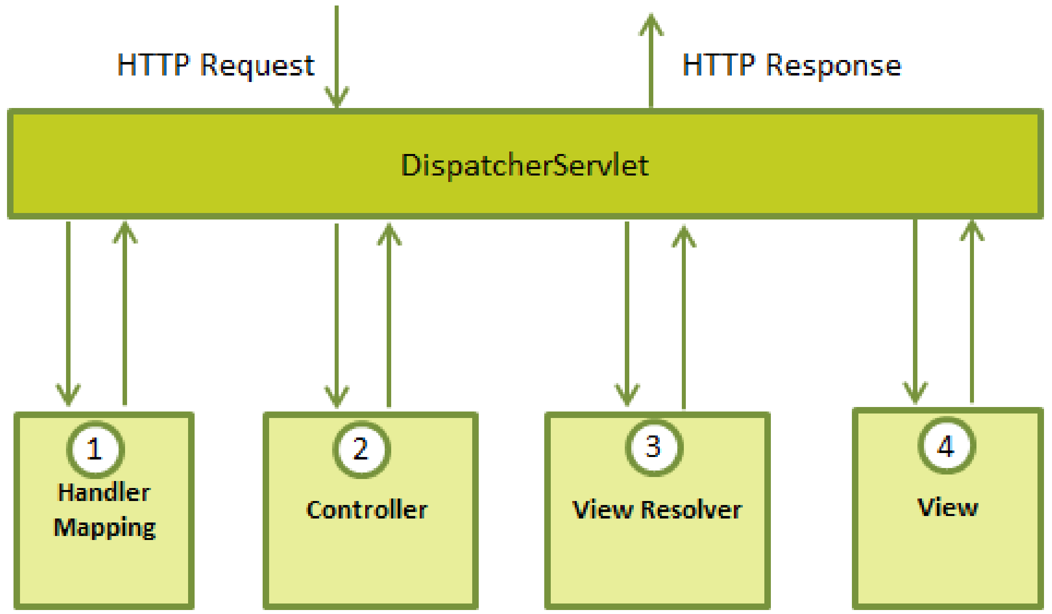
Spring MVC
MVC Bean Scopes

Scope	Description
Request	Scopes a single bean definition to the lifecycle of a single HTTP request; that is, each HTTP request has its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring ApplicationContext.
Session	Scopes a single bean definition to the lifecycle of an HTTP Session. Only valid in the context of a web-aware Spring ApplicationContext.
Global Session	Scopes a single bean definition to the lifecycle of a global HTTP Session. Typically only valid when used in a portlet context. Only valid in the context of a web-aware SpringApplicationContext.
Application	Scopes a single bean definition to the lifecycle of a ServletContext. Only valid in the context of a web-aware Spring ApplicationContext.

Spring MVC

The DispatcherServlet

- The Spring Web model-view-controller (MVC) framework is designed around a *DispatcherServlet* that handles all the HTTP requests and responses.



Spring MVC
Web.xml

- You need to map requests that you want the *DispatcherServlet* to handle, by using a URL mapping in the **web.xml** file.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>

<display-name>Example</display-name>

<context:component-scan base-package="com.proj.web" />

<context-param>
          <param-name>contextConfigLocation</param-name>
          <param-value>
          /WEB-INF/context/example-general-context.xml
          </param-value>
</context-param>

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

```
<servlet>
     <servlet-name>spring</servlet-name>
     <servlet-class>
     org.springframework.web.servlet.DispatcherServlet
     </servlet-class>
          <init-param>
          <param-name>contextConfigLocation</param-name>
          <param-value>/WEB-INF/example-servlet.xml</param-value>
          </init-param>
     <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
          <servlet-name>spring</servlet-name>
          <url-pattern>/</url-pattern>
</servlet-mapping>

</web-app>
```

Spring MVC

Servlet.xml

- Now, let us check the required configuration for **example-servlet.xml** file, placed in your web application's *WebContent/WEB-INF* directory:

```
<context:component-scan base-package="com.proj.web.controller" />

<bean id="viewResolver" class="org.springframework.web.servlet.view.UrlBasedViewResolver">
     <property name="viewClass" value="org.springframework.web.servlet.view.JstlView" />
     <property name="prefix" value="/WEB-INF/view/" />
     <property name="suffix" value=".jsp" />
</bean>

<mvc:annotation-driven />

</beans>
```

Spring MVC

Controller & View

- HomeController.java:

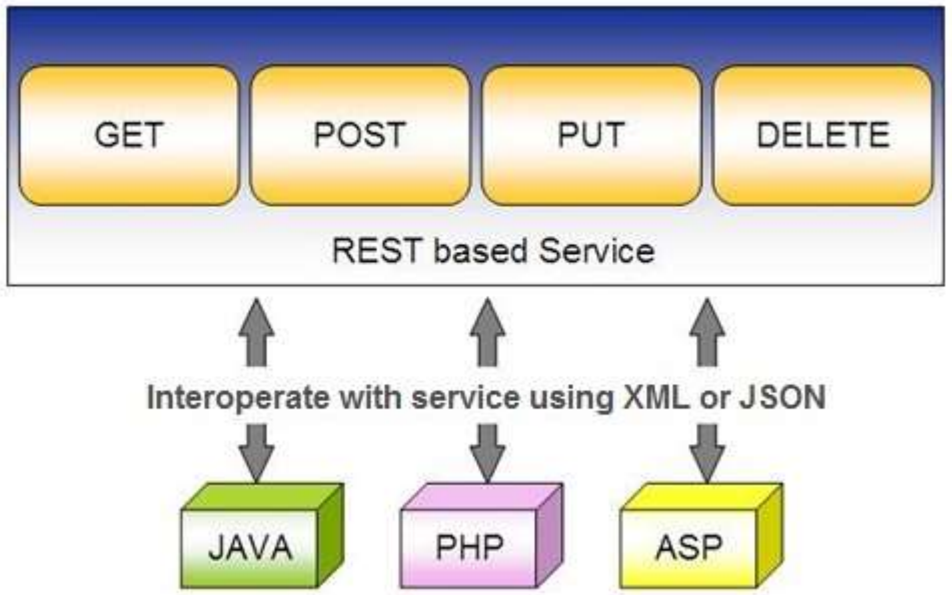
```
@Controller
public class HomeController {

@RequestMapping(value = { "/", "/home" },
method = RequestMethod.GET)

public String showHomePage(ModelMap model) {

     model.addAttribute("message", "Hello
     Spring MVC Framework!");

     return "home";
      }
}
```

- Home.jsp

```
<html>
<head>
<title>Here is home page</title>
</head>
<body>
<h2>${message}</h2>
</body>
</html>
```

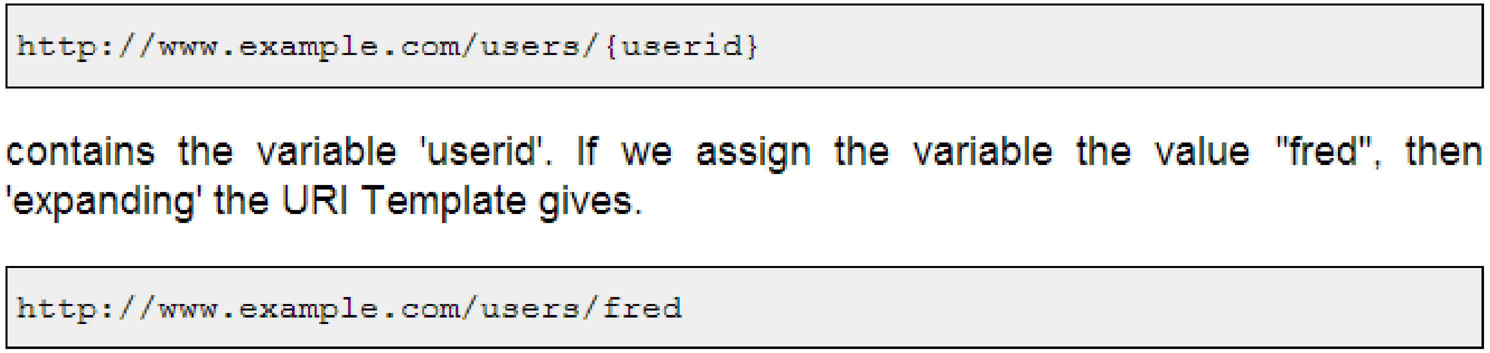
Spring RESTful Services

- *REST* does not require the client to know anything about the structure of the API. Rather, the server needs to provide whatever information the client needs to interact with the service.



Spring RESTful Services

- Spring's annotation-based MVC framework serves as the basis for creating RESTful Web Services.
- RESTful services use URIs to name resources. To facilitate accessing the information contained in a URI, its structure follows conventions so that it can easily be described in a parameterized form.



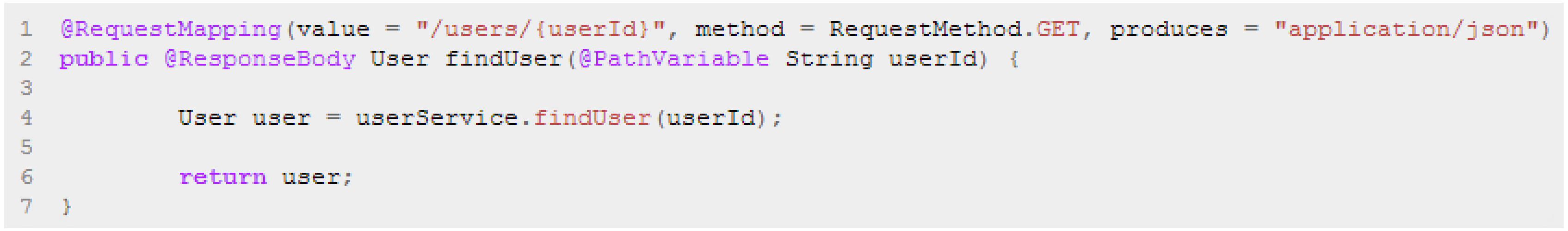
```
http://www.example.com/users/{userid}
```

contains the variable 'userid'. If we assign the variable the value "fred", then 'expanding' the URI Template gives.

```
http://www.example.com/users/fred
```

Spring RESTful Services

- Spring uses the @RequestMapping method annotation to define the URI Template for the request. The @PathVariable annotation is used to extract the value of the template variables and assign their value to a method variable.



```
@RequestMapping(value = "/users/{userId}", method = RequestMethod.GET, produces = "application/json")
public @ResponseBody User findUser(@PathVariable String userId) {

        User user = userService.findUser(userId);

        return user;
}
```

- When a request comes in for */users/serhat*, the value *'serhat'* is bound to the method parameter String *userId*.

Spring RESTful Services

- A RESTful architecture may expose multiple representations of a resource. There are two strategies for a client to inform the server of the representation it is interested in receiving.

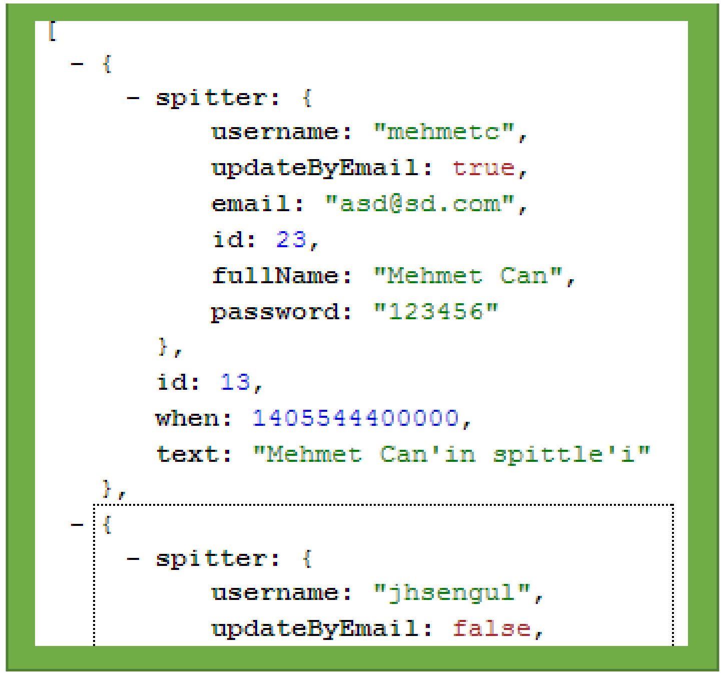
- The first strategy is to use a distinct URI for each resource. This is typically done by using a different file extension in the URI.
    - For example the URI http://www.example.com/users/serhat.pdf requests a PDF representation of the user serhat while http://www.example.com/users/serhat.xml requests an XML representation.
- The second strategy is for the client to use the same URI to locate the resource but set the Accept HTTP request header to list the media types that it understands.
    - For example, a HTTP request for http://www.example.com/users/serhat with an Accept header set to *application/pdf* requests a PDF representation of the user serhat while http://www.example.com/users/serhat with an Accept header set to *text/xml* requests an XML representation. This strategy is known as content negotiation.

Spring RESTful Services

```java
@RequestMapping(value = "/restfuldeneme", method = RequestMethod.GET, headers = "Accept=application/json")
public @ResponseBody List<Spittle> mymethod() {
    return spitterService.getRecentSpittles(spittlesPerPage);
}
```

```java
@Entity
@Table(name = "spitter")
@XmlRootElement
public class Spitter implements Serializable {
    private static final long serialVersionUID = 1L;
```

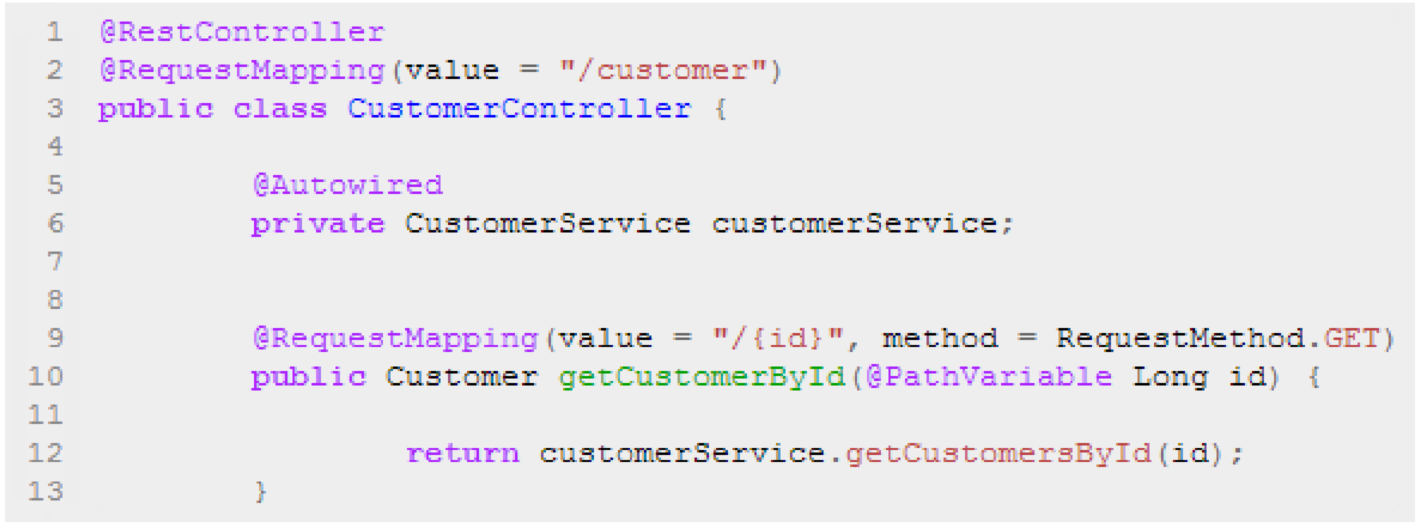
Do not forget to implement serializable in your entity



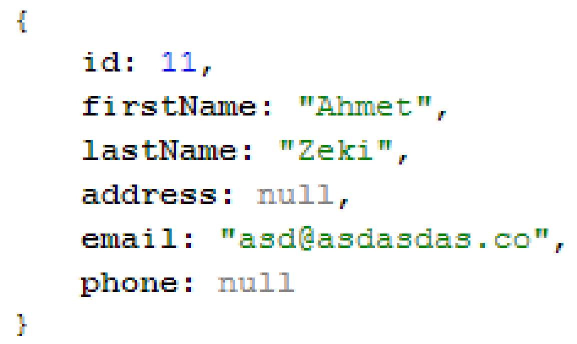
```
[
  - {
      - spitter: {
            username: "mehmetc",
            updateByEmail: true,
            email: "asd@sd.com",
            id: 23,
            fullName: "Mehmet Can",
            password: "123456"
        },
        id: 13,
        when: 1405544400000,
        text: "Mehmet Can'in spittle'i"
    },
  - {
      - spitter: {
            username: "jhsengul",
            updateByEmail: false,
```

Spring RESTful Services

- This code uses Spring 4’s new @RestController annotation, which marks the class as a controller where every method returns a domain object instead of a view. It’s shorthand for @Controller and @ResponseBody rolled together.
- The Greeting object must be converted to JSON. Thanks to Spring’s HTTP message converter support, you don’t need to do this conversion manually.



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

        @Autowired
        private CustomerService customerService;


        @RequestMapping(value = "/{id}", method = RequestMethod.GET)
        public Customer getCustomerById(@PathVariable Long id) {

                return customerService.getCustomersById(id);
        }
```



```
{
    id: 11,
    firstName: "Ahmet",
    lastName: "Zeki",
    address: null,
    email: "asd@asdasdas.co",
    phone: null
}
```

Spring Security

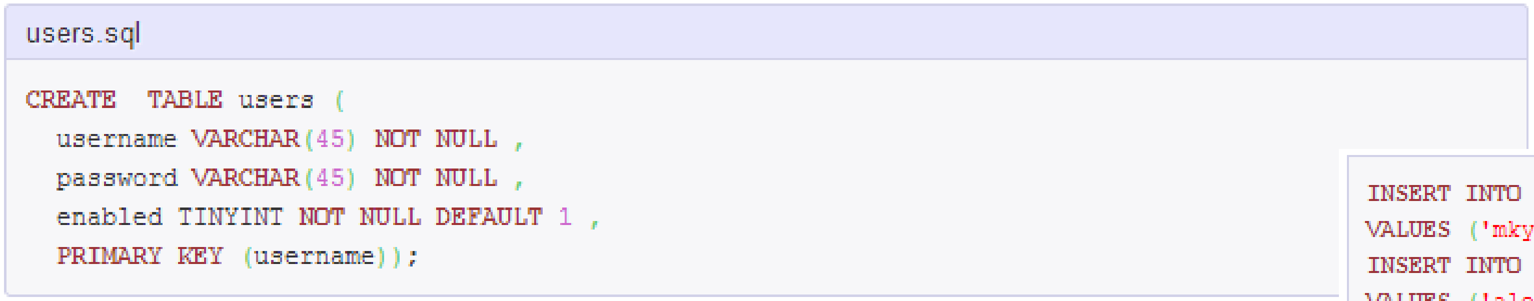
Spring Security is a framework that focuses on providing both authentication and authorization to Java applications.

Features:

- Comprehensive and extensible support for both Authentication and Authorization
- Protection against attacks like session fixation, clickjacking, cross site request forgery, etc
- Servlet API integration
- Optional integration with Spring Web MVC
- Much more...

Spring Security
Authentication & Authorization

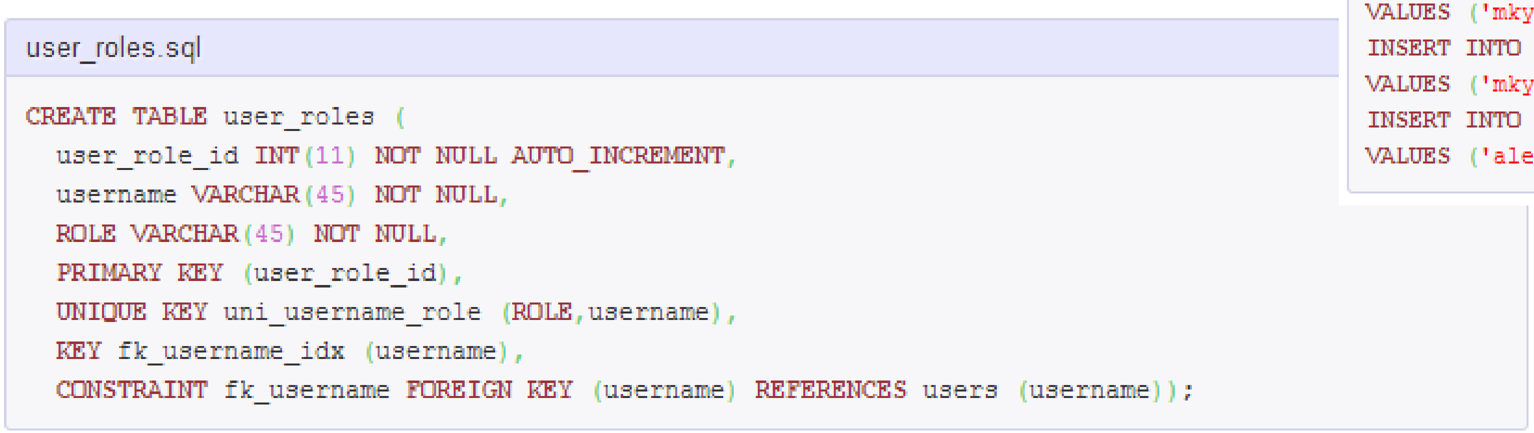
4.1 Create a "users" table.



users.sql

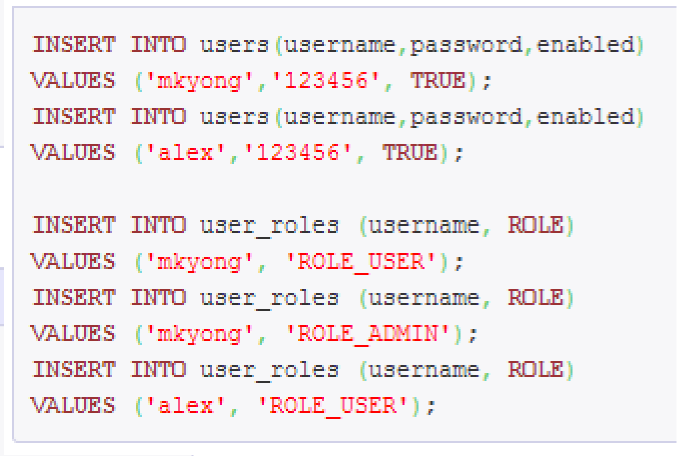
```
CREATE  TABLE users (
  username VARCHAR(45) NOT NULL ,
  password VARCHAR(45) NOT NULL ,
  enabled TINYINT NOT NULL DEFAULT 1 ,
  PRIMARY KEY (username));
```

4.2 Create a "user_roles" table.



user_roles.sql

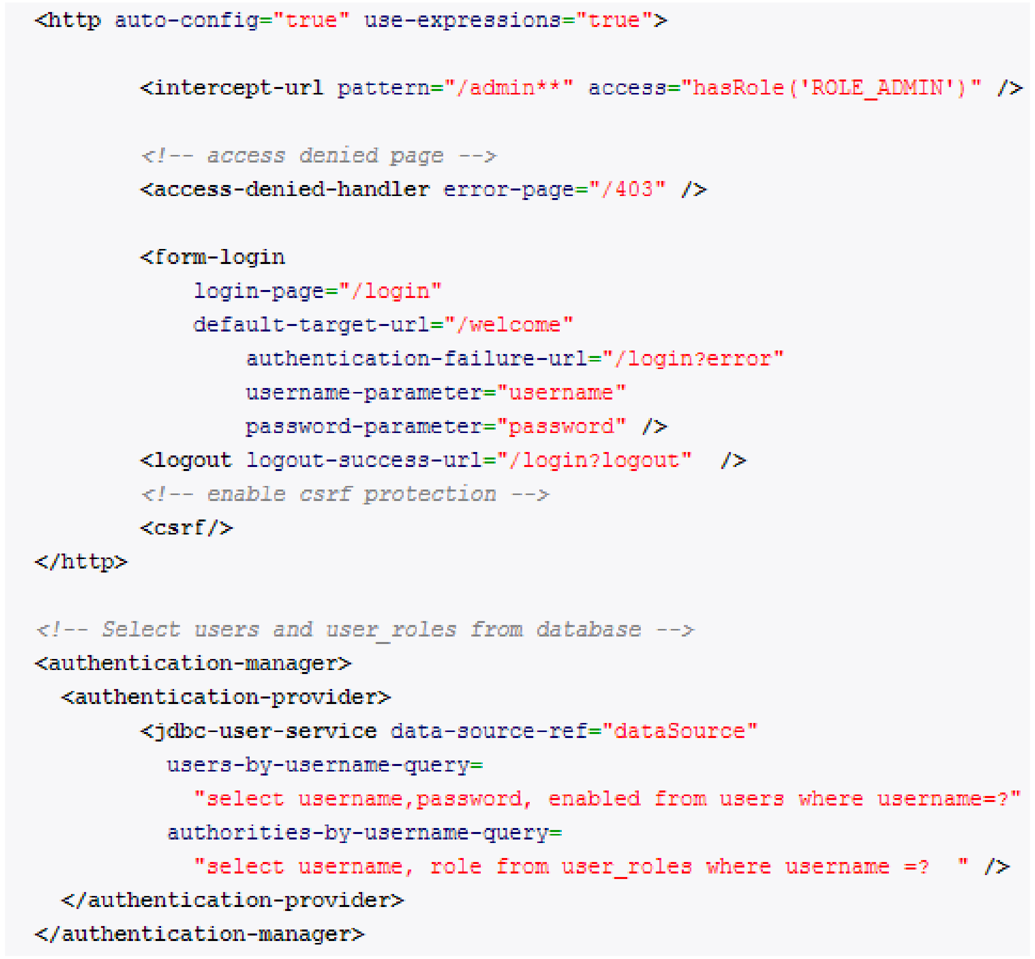
```
CREATE TABLE user_roles (
  user_role_id INT(11) NOT NULL AUTO_INCREMENT,
  username VARCHAR(45) NOT NULL,
  ROLE VARCHAR(45) NOT NULL,
  PRIMARY KEY (user_role_id),
  UNIQUE KEY uni_username_role (ROLE,username),
  KEY fk_username_idx (username),
  CONSTRAINT fk_username FOREIGN KEY (username) REFERENCES users (username));
```



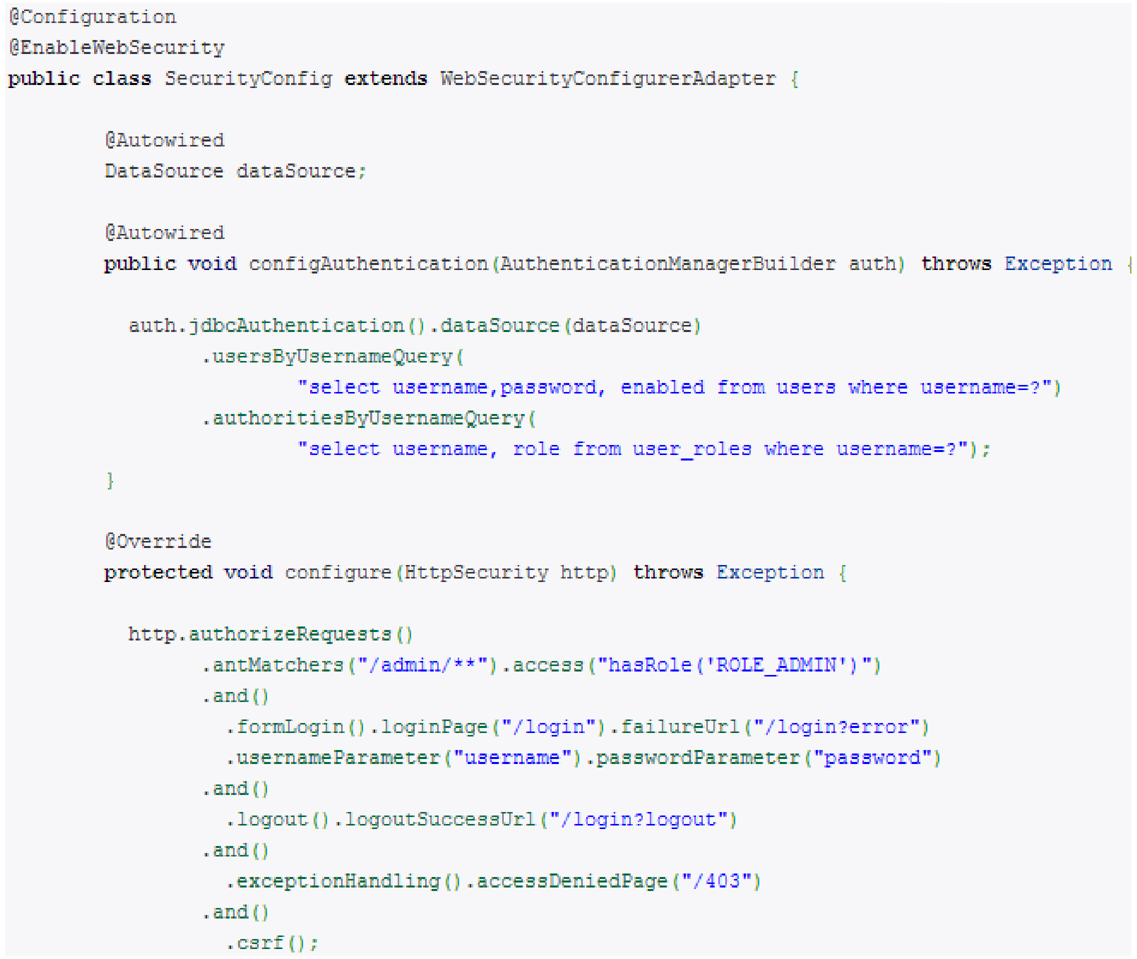
```
INSERT INTO users(username,password,enabled)
VALUES ('mkyong','123456', TRUE);
INSERT INTO users(username,password,enabled)
VALUES ('alex','123456', TRUE);

INSERT INTO user_roles (username, ROLE)
VALUES ('mkyong', 'ROLE_USER');
INSERT INTO user_roles (username, ROLE)
VALUES ('mkyong', 'ROLE_ADMIN');
INSERT INTO user_roles (username, ROLE)
VALUES ('alex', 'ROLE_USER');
```

Spring Security

Authentication & Authorization



```xml
<http auto-config="true" use-expressions="true">

        <intercept-url pattern="/admin**" access="hasRole('ROLE_ADMIN')" />

        <!-- access denied page -->
        <access-denied-handler error-page="/403" />

        <form-login
            login-page="/login"
            default-target-url="/welcome"
                authentication-failure-url="/login?error"
                username-parameter="username"
                password-parameter="password" />
        <logout logout-success-url="/login?logout"  />
        <!-- enable csrf protection -->
        <csrf/>
</http>

<!-- Select users and user_roles from database -->
<authentication-manager>
  <authentication-provider>
        <jdbc-user-service data-source-ref="dataSource"
          users-by-username-query=
            "select username,password, enabled from users where username=?"
          authorities-by-username-query=
            "select username, role from user_roles where username =?  " />
  </authentication-provider>
</authentication-manager>
```



```java
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

        @Autowired
        DataSource dataSource;

        @Autowired
        public void configAuthentication(AuthenticationManagerBuilder auth) throws Exception {

          auth.jdbcAuthentication().dataSource(dataSource)
                .usersByUsernameQuery(
                        "select username,password, enabled from users where username=?")
                .authoritiesByUsernameQuery(
                        "select username, role from user_roles where username=?");
        }

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

          http.authorizeRequests()
                .antMatchers("/admin/**").access("hasRole('ROLE_ADMIN')")
                .and()
                  .formLogin().loginPage("/login").failureUrl("/login?error")
                  .usernameParameter("username").passwordParameter("password")
                .and()
                  .logout().logoutSuccessUrl("/login?logout")
                .and()
                  .exceptionHandling().accessDeniedPage("/403")
                .and()
                  .csrf();
```

Spring Security

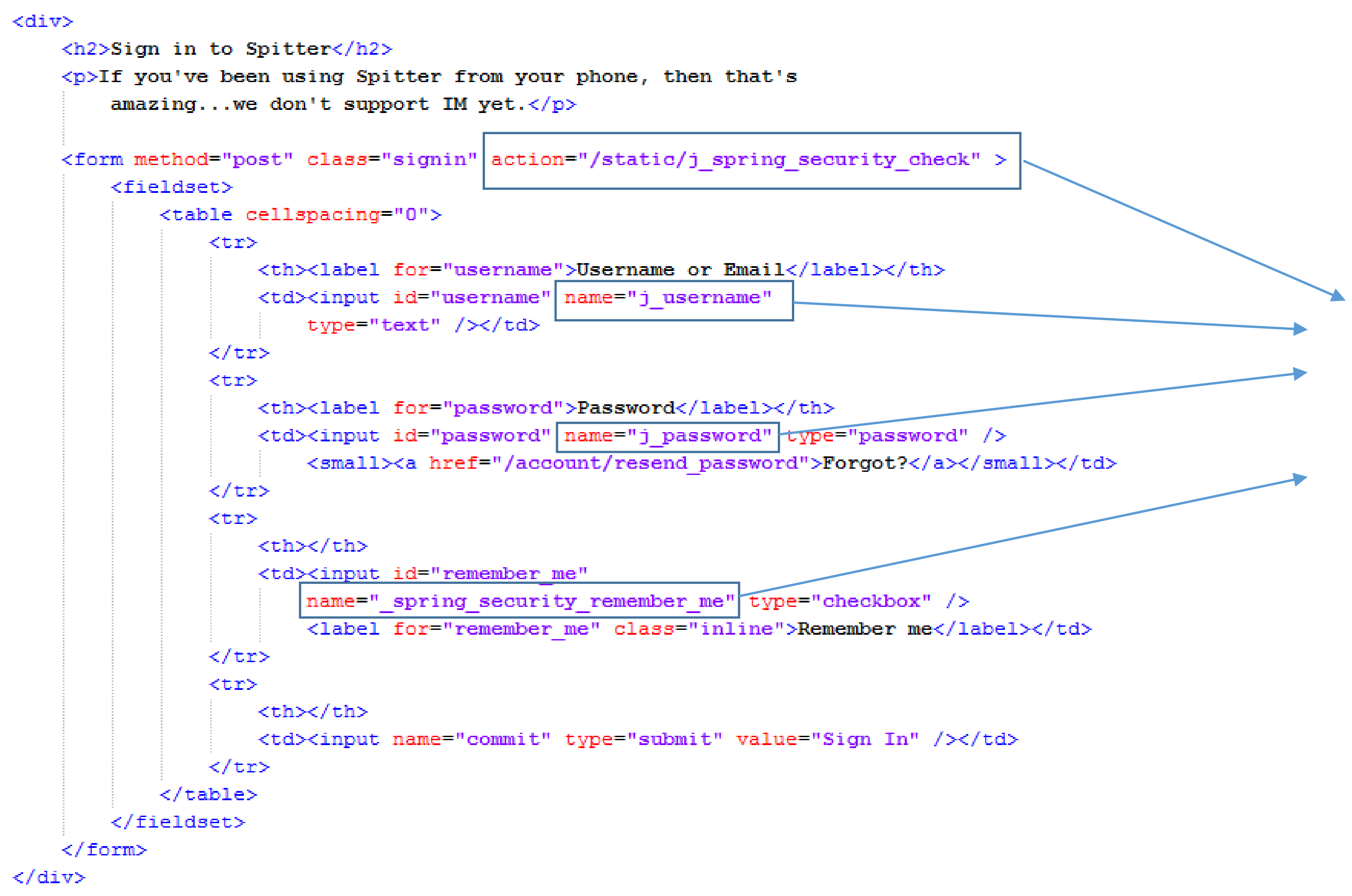
Authentication & Authorization

```
<http auto-config="true" use-expressions="true">
    <form-login login-page="/login" login-processing-url="/static/j_spring_security_check"
        authentication-failure-url="/login?error" />
    <logout logout-url="/static/j_spring_security_logout" />
    <intercept-url pattern="/login" access="permitAll" />
    <intercept-url pattern="/home" access="hasRole('ROLE_SPITTER')" />
    <intercept-url pattern="/admin/**"
        access="isAuthenticated() and principal.username=='cansrht'" />

    <intercept-url pattern="/login" requires-channel="https" />
    <intercept-url pattern="/spitter/form" requires-channel="https" />

    <remember-me key="spitterKey" token-validity-seconds="2419200" />

</http>
```

Lastly, forcing application to use secure channel (https) is easy to implement in Spring Security.

Lastly, forcing application to use secure channel (https) is easy to implement in Spring Security.

Spring Security
Authentication & Authorization



```
<div>
    <h2>Sign in to Spitter</h2>
    <p>If you've been using Spitter from your phone, then that's
        amazing...we don't support IM yet.</p>

    <form method="post" class="signin" action="/static/j_spring_security_check" >
        <fieldset>
            <table cellspacing="0">
                <tr>
                    <th><label for="username">Username or Email</label></th>
                    <td><input id="username" name="j_username"
                        type="text" /></td>
                </tr>
                <tr>
                    <th><label for="password">Password</label></th>
                    <td><input id="password" name="j_password" type="password" />
                        <small><a href="/account/resend_password">Forgot?</a></small></td>
                </tr>
                <tr>
                    <th></th>
                    <td><input id="remember_me"
                        name="_spring_security_remember_me" type="checkbox" />
                        <label for="remember_me" class="inline">Remember me</label></td>
                </tr>
                <tr>
                    <th></th>
                    <td><input name="commit" type="submit" value="Sign In" /></td>
                </tr>
            </table>
        </fieldset>
    </form>
</div>
```

Spring special naming for

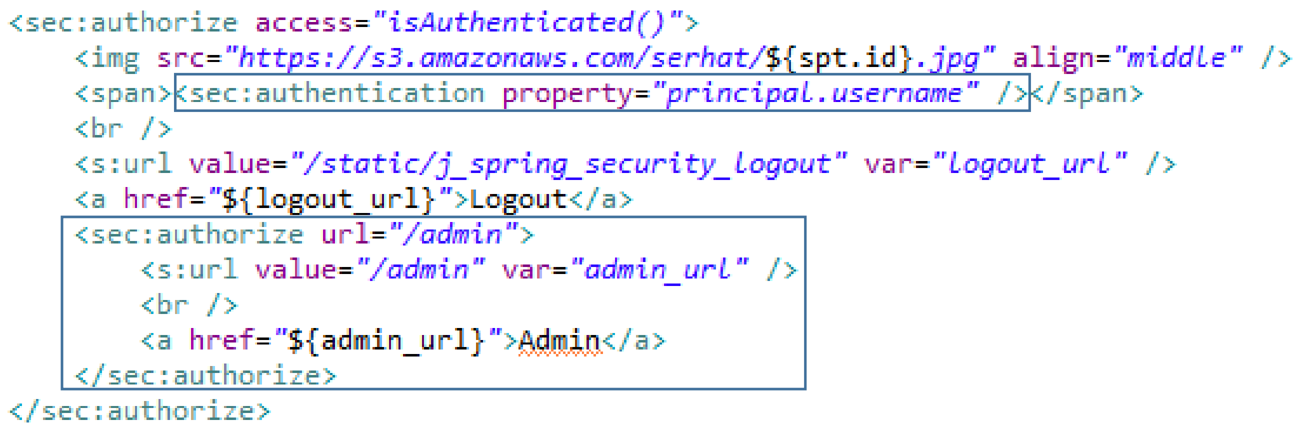
Spring supported Authentication

Spring Security

Authentication & Authorization

- The authorize & authentication tag

```
<sec:authorize access="hasRole('supervisor')">

This content will only be visible to users who have
the "supervisor" authority in their list of <tt>GrantedAuthority</tt>s.

</sec:authorize>
```

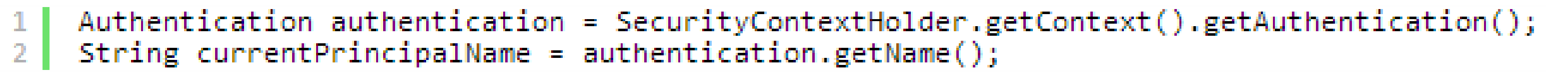


```
<sec:authorize access="isAuthenticated()">
    <img src="https://s3.amazonaws.com/serhat/${spt.id}.jpg" align="middle" />
    <span><sec:authentication property="principal.username" /></span>
    <br />
    <s:url value="/static/j_spring_security_logout" var="logout_url" />
    <a href="${logout_url}">Logout</a>
    <sec:authorize url="/admin">
        <s:url value="/admin" var="admin_url" />
        <br />
        <a href="${admin_url}">Admin</a>
    </sec:authorize>
</sec:authorize>
```

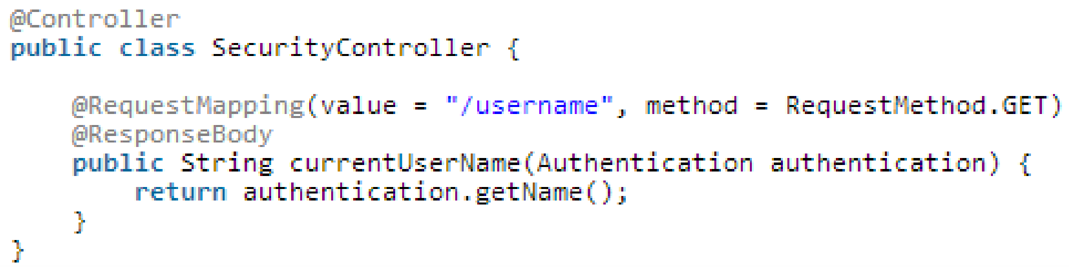
Spring Security

Authentication & Authorization

- You can access the Authentication object in your MVC controller (by calling SecurityContextHolder.getContext().getAuthentication()) and add the data directly to your model for rendering by the view.



```java
Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
String currentPrincipalName = authentication.getName();
```

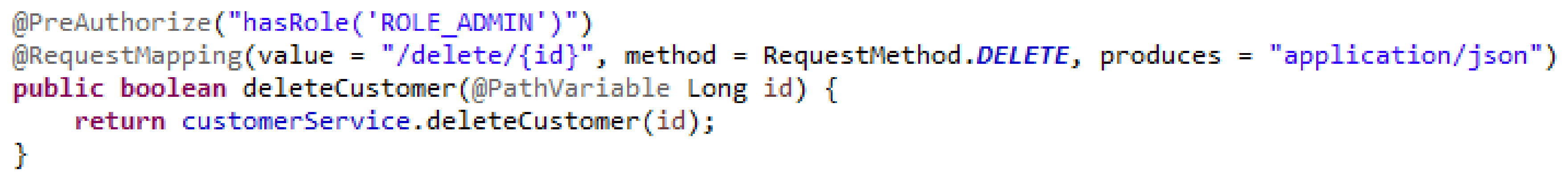


```java
@Controller
public class SecurityController {

    @RequestMapping(value = "/username", method = RequestMethod.GET)
    @ResponseBody
    public String currentUserName(Authentication authentication) {
        return authentication.getName();
    }
}
```

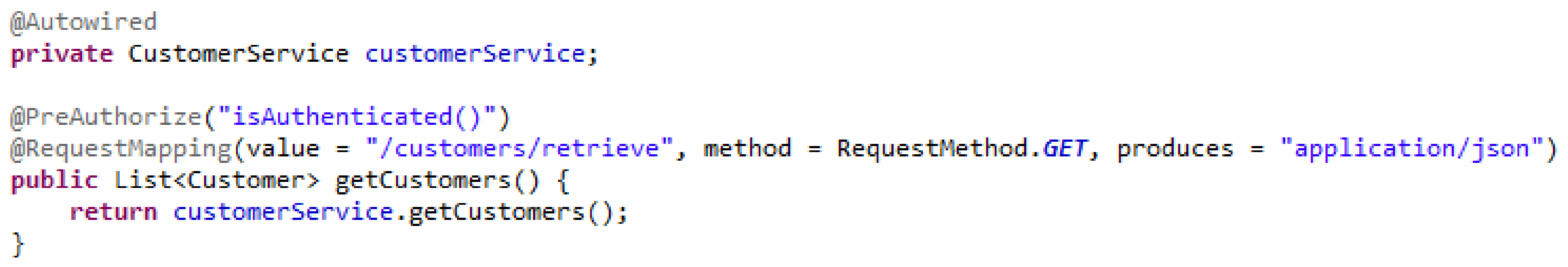
Spring Security

Authentication & Authorization

- Authorization with annotations in RESTful Web Service



```
@PreAuthorize("hasRole('ROLE_ADMIN')")
@RequestMapping(value = "/delete/{id}", method = RequestMethod.DELETE, produces = "application/json")
public boolean deleteCustomer(@PathVariable Long id) {
    return customerService.deleteCustomer(id);
}
```



```
@Autowired
private CustomerService customerService;

@PreAuthorize("isAuthenticated()")
@RequestMapping(value = "/customers/retrieve", method = RequestMethod.GET, produces = "application/json")
public List<Customer> getCustomers() {
    return customerService.getCustomers();
}
```

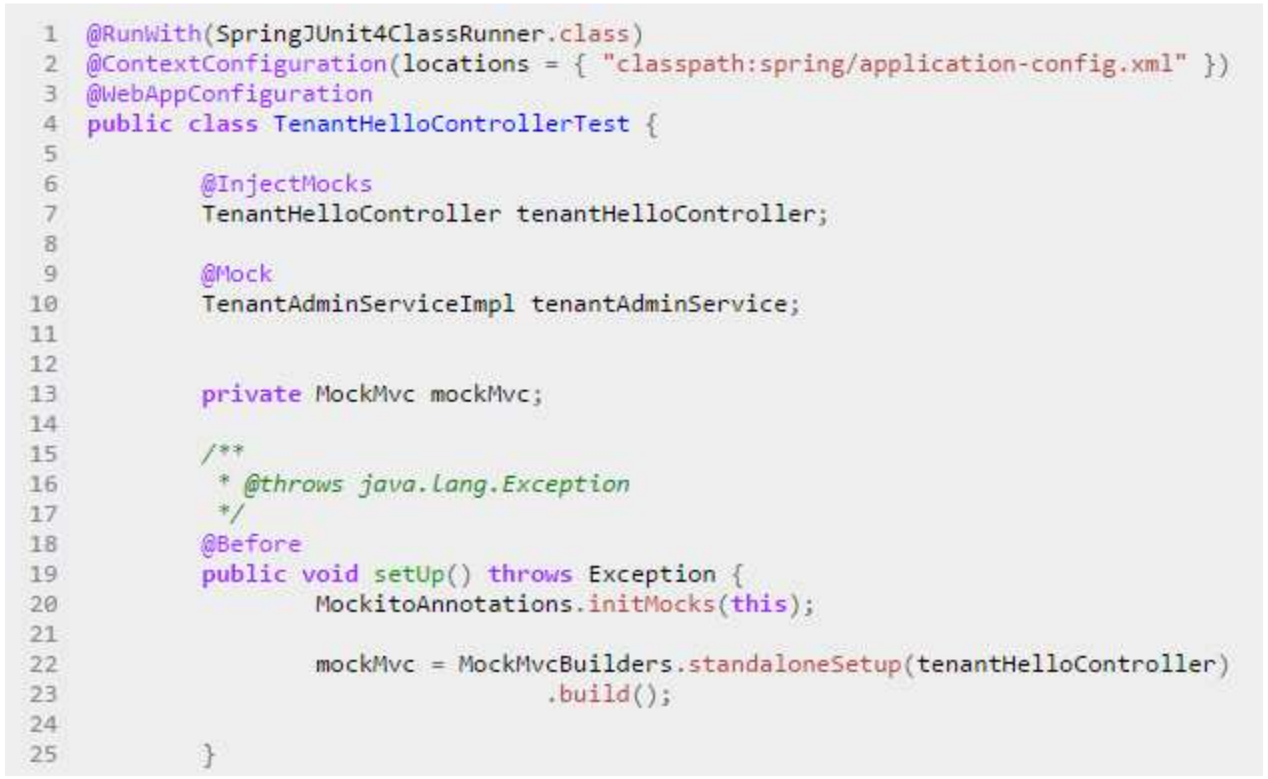
Spring Test

Spring Test Framework supports;

- Unit testing with mock objects
- Easy unit testing for Controllers
- IoC container to create dependencies for Integration Testing
- Transaction management for Integration Testing
- Third party frameworks like JUnit, TestNG, Mockito

Spring Test

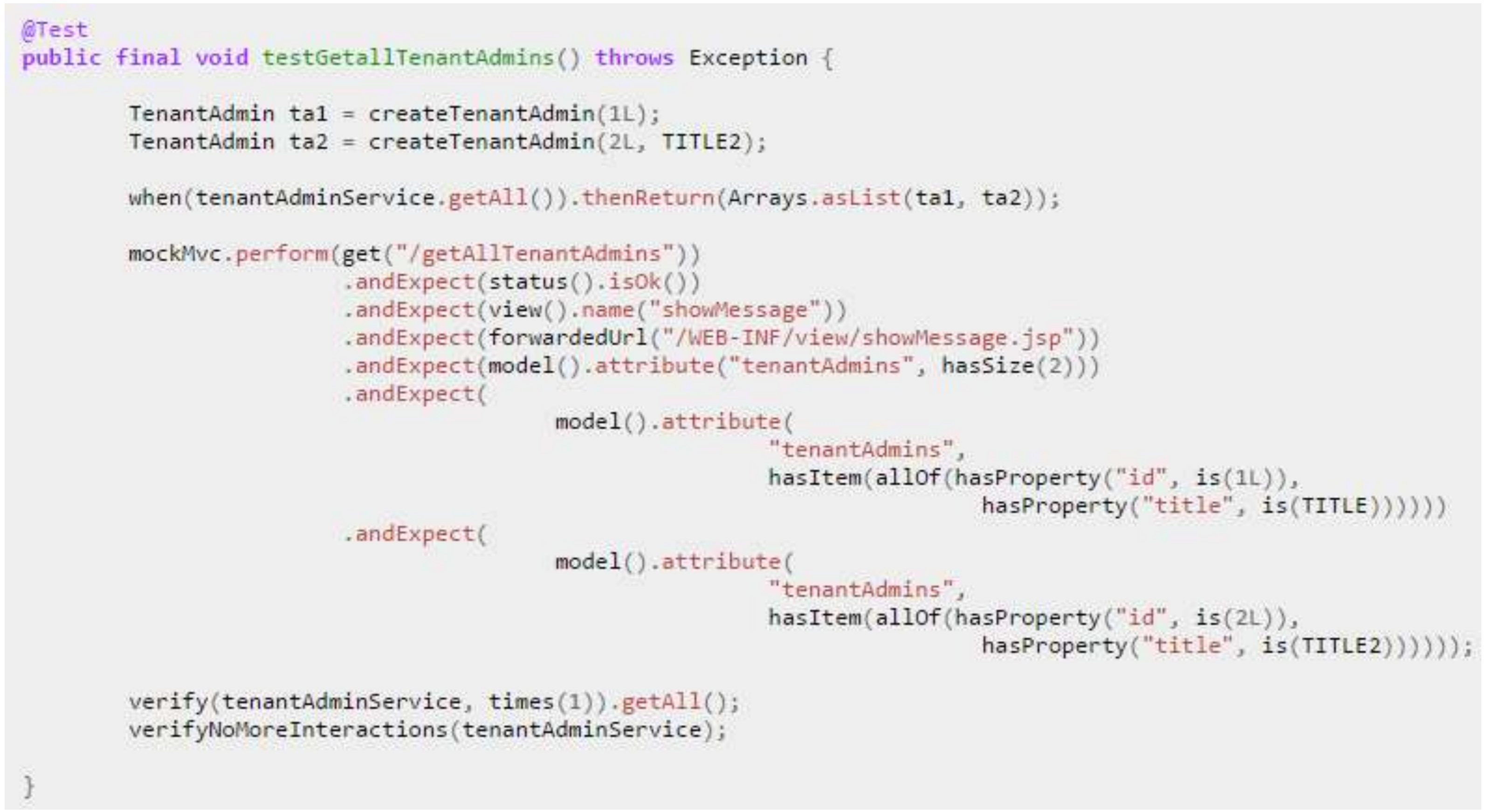
Unit Testing



```java
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = { "classpath:spring/application-config.xml" })
@WebAppConfiguration
public class TenantHelloControllerTest {

        @InjectMocks
        TenantHelloController tenantHelloController;

        @Mock
        TenantAdminServiceImpl tenantAdminService;


        private MockMvc mockMvc;

        /**
         * @throws java.lang.Exception
         */
        @Before
        public void setUp() throws Exception {
                MockitoAnnotations.initMocks(this);

                mockMvc = MockMvcBuilders.standaloneSetup(tenantHelloController)
                                .build();

        }
```

Spring Test

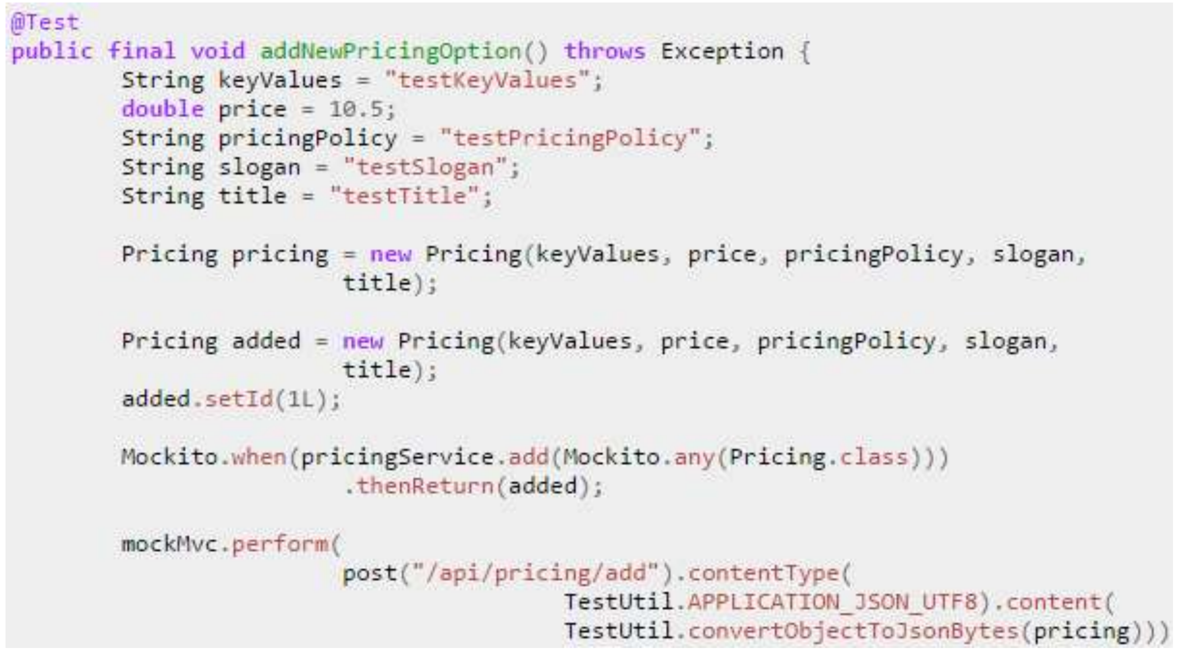
Unit Testing



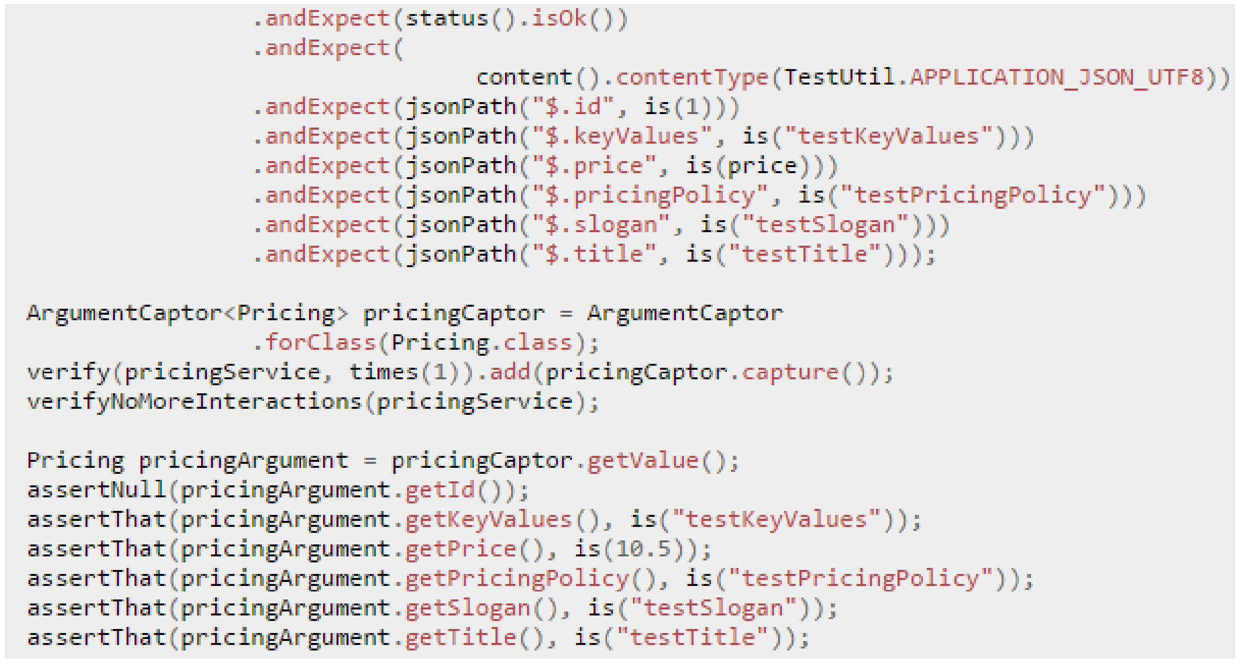
```
@Test
public final void testGetallTenantAdmins() throws Exception {

        TenantAdmin ta1 = createTenantAdmin(1L);
        TenantAdmin ta2 = createTenantAdmin(2L, TITLE2);

        when(tenantAdminService.getAll()).thenReturn(Arrays.asList(ta1, ta2));

        mockMvc.perform(get("/getAllTenantAdmins"))
                        .andExpect(status().isOk())
                        .andExpect(view().name("showMessage"))
                        .andExpect(forwardedUrl("/WEB-INF/view/showMessage.jsp"))
                        .andExpect(model().attribute("tenantAdmins", hasSize(2)))
                        .andExpect(
                                model().attribute(
                                        "tenantAdmins",
                                        hasItem(allOf(hasProperty("id", is(1L)),
                                                hasProperty("title", is(TITLE))))))
                        .andExpect(
                                model().attribute(
                                        "tenantAdmins",
                                        hasItem(allOf(hasProperty("id", is(2L)),
                                                hasProperty("title", is(TITLE2))))));

        verify(tenantAdminService, times(1)).getAll();
        verifyNoMoreInteractions(tenantAdminService);

}
```

Spring Test

Unit Testing

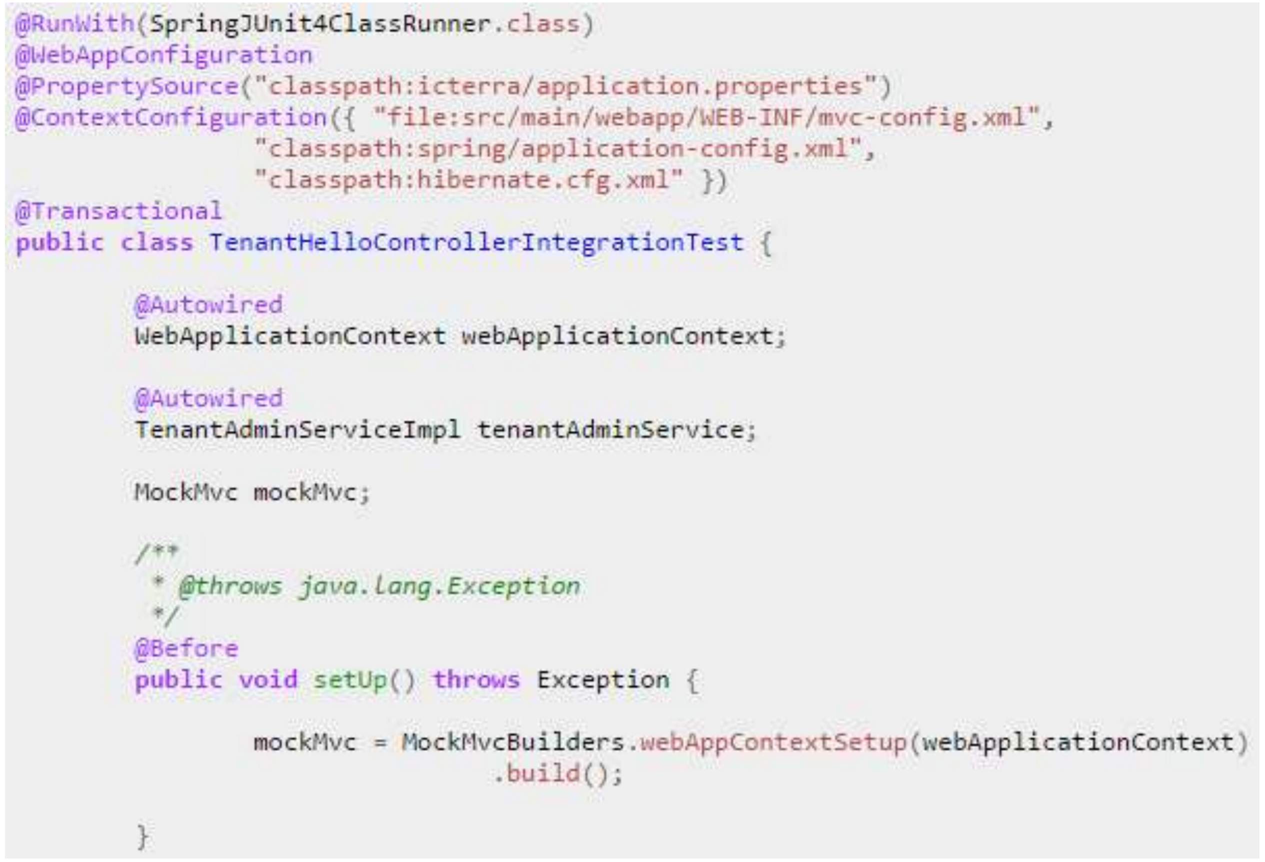


```java
@Test
public final void addNewPricingOption() throws Exception {
        String keyValues = "testKeyValues";
        double price = 10.5;
        String pricingPolicy = "testPricingPolicy";
        String slogan = "testSlogan";
        String title = "testTitle";

        Pricing pricing = new Pricing(keyValues, price, pricingPolicy, slogan,
                        title);

        Pricing added = new Pricing(keyValues, price, pricingPolicy, slogan,
                        title);
        added.setId(1L);

        Mockito.when(pricingService.add(Mockito.any(Pricing.class)))
                        .thenReturn(added);

        mockMvc.perform(
                        post("/api/pricing/add").contentType(
                                        TestUtil.APPLICATION_JSON_UTF8).content(
                                        TestUtil.convertObjectToJsonBytes(pricing)))
```



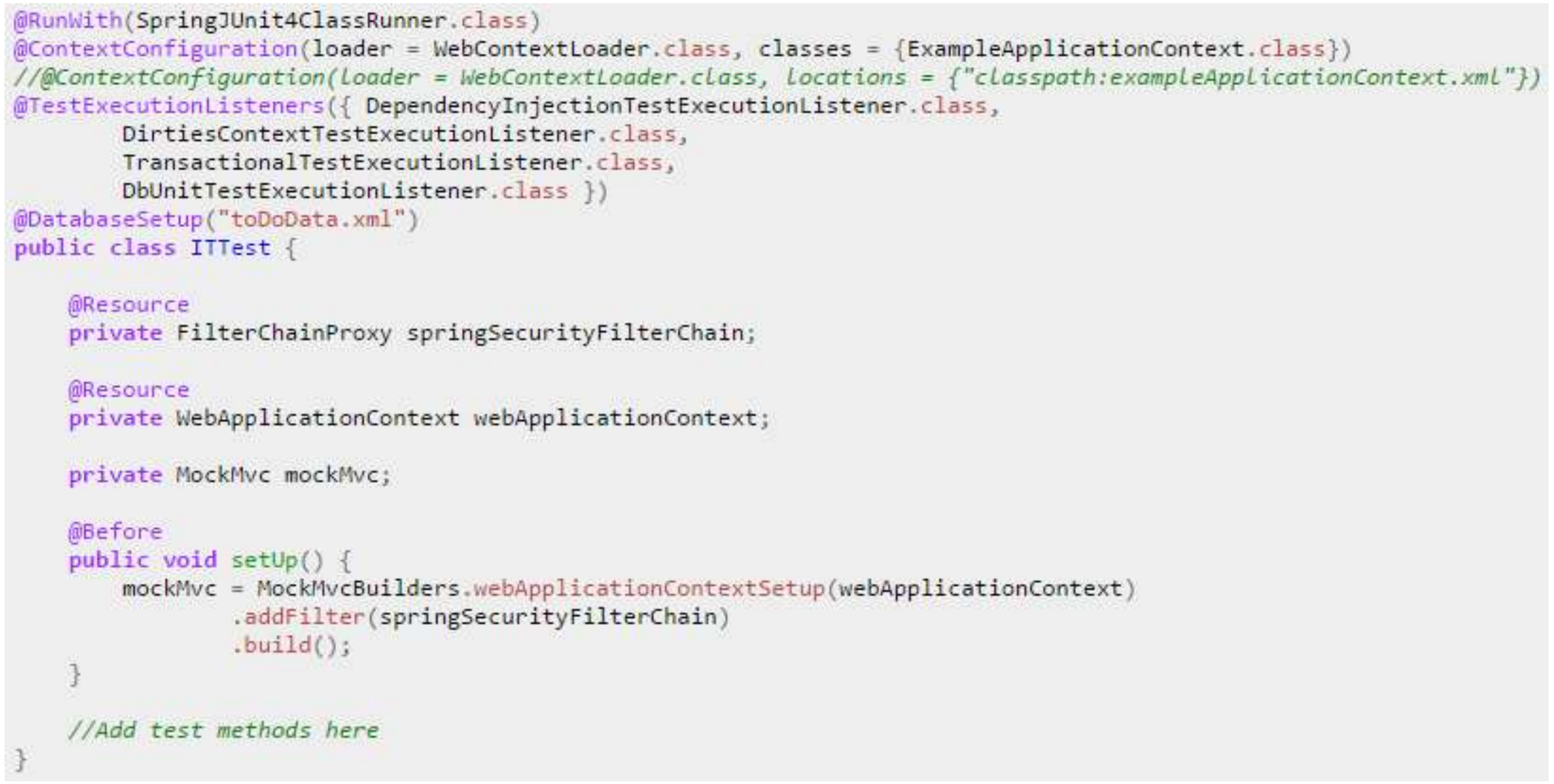
```java
                        .andExpect(status().isOk())
                        .andExpect(
                                        content().contentType(TestUtil.APPLICATION_JSON_UTF8))
                        .andExpect(jsonPath("$.id", is(1)))
                        .andExpect(jsonPath("$.keyValues", is("testKeyValues")))
                        .andExpect(jsonPath("$.price", is(price)))
                        .andExpect(jsonPath("$.pricingPolicy", is("testPricingPolicy")))
                        .andExpect(jsonPath("$.slogan", is("testSlogan")))
                        .andExpect(jsonPath("$.title", is("testTitle")));

ArgumentCaptor<Pricing> pricingCaptor = ArgumentCaptor
                        .forClass(Pricing.class);
verify(pricingService, times(1)).add(pricingCaptor.capture());
verifyNoMoreInteractions(pricingService);

Pricing pricingArgument = pricingCaptor.getValue();
assertNull(pricingArgument.getId());
assertThat(pricingArgument.getKeyValues(), is("testKeyValues"));
assertThat(pricingArgument.getPrice(), is(10.5));
assertThat(pricingArgument.getPricingPolicy(), is("testPricingPolicy"));
assertThat(pricingArgument.getSlogan(), is("testSlogan"));
assertThat(pricingArgument.getTitle(), is("testTitle"));
```

Spring Test
Integration Testing



```
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@PropertySource("classpath:icterra/application.properties")
@ContextConfiguration({ "file:src/main/webapp/WEB-INF/mvc-config.xml",
                "classpath:spring/application-config.xml",
                "classpath:hibernate.cfg.xml" })
@Transactional
public class TenantHelloControllerIntegrationTest {

        @Autowired
        WebApplicationContext webApplicationContext;

        @Autowired
        TenantAdminServiceImpl tenantAdminService;

        MockMvc mockMvc;

        /**
         * @throws java.lang.Exception
         */
        @Before
        public void setUp() throws Exception {

                mockMvc = MockMvcBuilders.webAppContextSetup(webApplicationContext)
                                .build();

        }
```

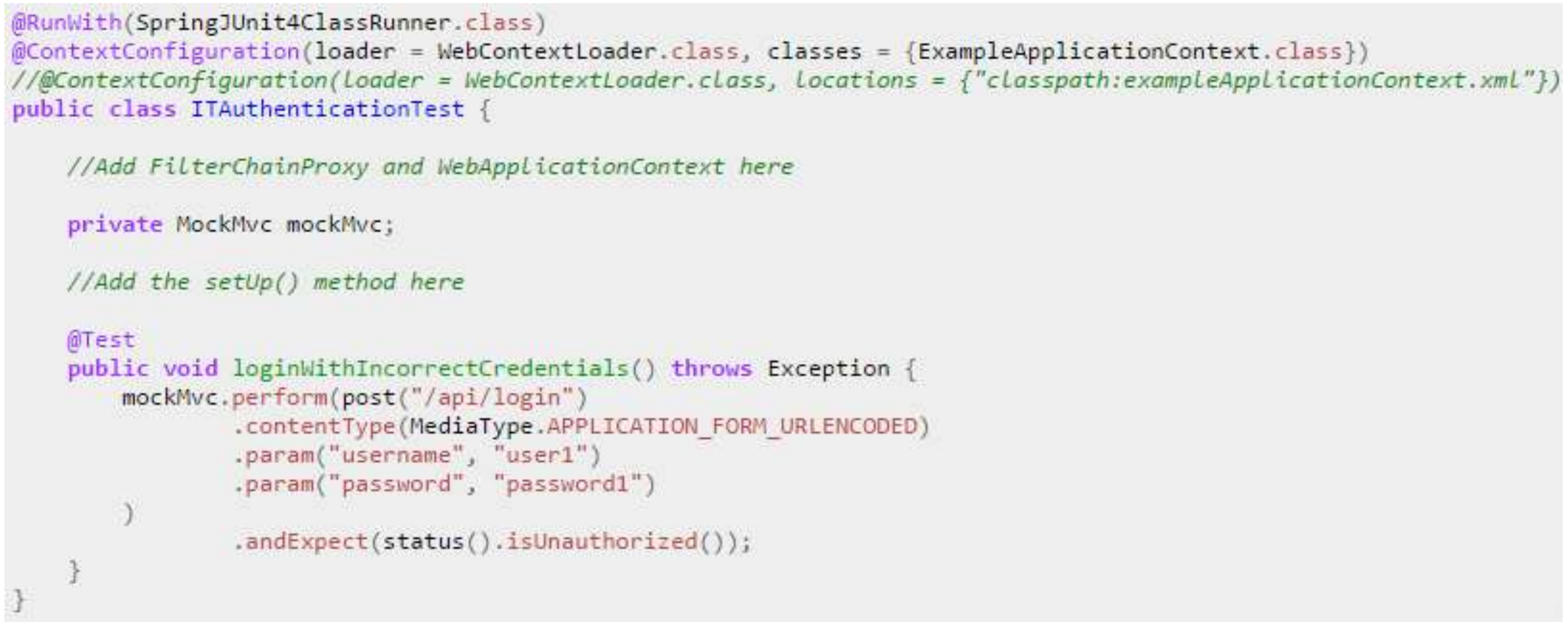
Spring Test
Integration Testing



```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(loader = WebContextLoader.class, classes = {ExampleApplicationContext.class})
//@ContextConfiguration(loader = WebContextLoader.class, locations = {"classpath:exampleApplicationContext.xml"})
@TestExecutionListeners({ DependencyInjectionTestExecutionListener.class,
        DirtiesContextTestExecutionListener.class,
        TransactionalTestExecutionListener.class,
        DbUnitTestExecutionListener.class })
@DatabaseSetup("toDoData.xml")
public class ITTest {

    @Resource
    private FilterChainProxy springSecurityFilterChain;

    @Resource
    private WebApplicationContext webApplicationContext;

    private MockMvc mockMvc;

    @Before
    public void setUp() {
        mockMvc = MockMvcBuilders.webApplicationContextSetup(webApplicationContext)
                .addFilter(springSecurityFilterChain)
                .build();
    }

    //Add test methods here
}
```

Spring Test

Integration Testing



```java
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(loader = WebContextLoader.class, classes = {ExampleApplicationContext.class})
//@ContextConfiguration(loader = WebContextLoader.class, locations = {"classpath:exampleApplicationContext.xml"})
public class ITAuthenticationTest {

    //Add FilterChainProxy and WebApplicationContext here

    private MockMvc mockMvc;

    //Add the setUp() method here

    @Test
    public void loginWithIncorrectCredentials() throws Exception {
        mockMvc.perform(post("/api/login")
                .contentType(MediaType.APPLICATION_FORM_URLENCODED)
                .param("username", "user1")
                .param("password", "password1")
        )
                .andExpect(status().isUnauthorized());
    }
}
```

References

- http://docs.spring.io/spring/docs/current/spring-framework-reference/html/
- http://projects.spring.io/spring-security/
- http://www.mkyong.com/tutorials/spring-mvc-tutorials/
- http://www.mkyong.com/tutorials/spring-security-tutorials/
- http://www.tutorialspoint.com/spring/
- http://www.mkyong.com/tutorials/spring-tutorials/
- http://www.slideshare.net/rstoya05/testing-web-apps-with-spring-framework-32
- http://www.petrikainulainen.net/programming/spring-framework/integration-testing-of-spring-mvc-applications-security/

Thank you for listening…