WebSystique

learn together

Spring @Profile Example

August 20, 2014 websystiqueadmin

In this tutorial we will explore Spring `@Profile` annotation and use it to make different set of beans or configuration available conditionally on different environments. We will also discuss corresponding XML based profile configuration for comparison.

Let's imagine you have an application which includes database-interaction. You might want to configure one dateSource (for MySQL e.g.) for Development environment while completely different dataSource (for ORACLE e.g.) for Production.

Using `Spring Profiles`, you can easily manage such setup. We will explore this example usecase in this post.

Other interesting posts you may like

- Spring MVC 4+AngularJS Example
- Spring MVC 4+Hibernate 4 Many-to-many JSP Example
- Spring MVC 4+Hibernate 4+MySQL+Maven integration example using annotations
- Spring MVC4 FileUpload-Download Hibernate+MySQL Example
- TestNG Mockito Integration Example Stubbing Void Methods
- Maven surefire plugin and TestNG Example

Recent Posts

Spring 4 MVC+AngularJS CRUD Application using ngResource

Angularjs Server Communication using ngResource-CRUD Application

AngularJS Custom-Directives controllers, require option guide

AngularJS Custom-Directives transclude, ngTransclude guide

AngularJS Custom-Directives replace option guide

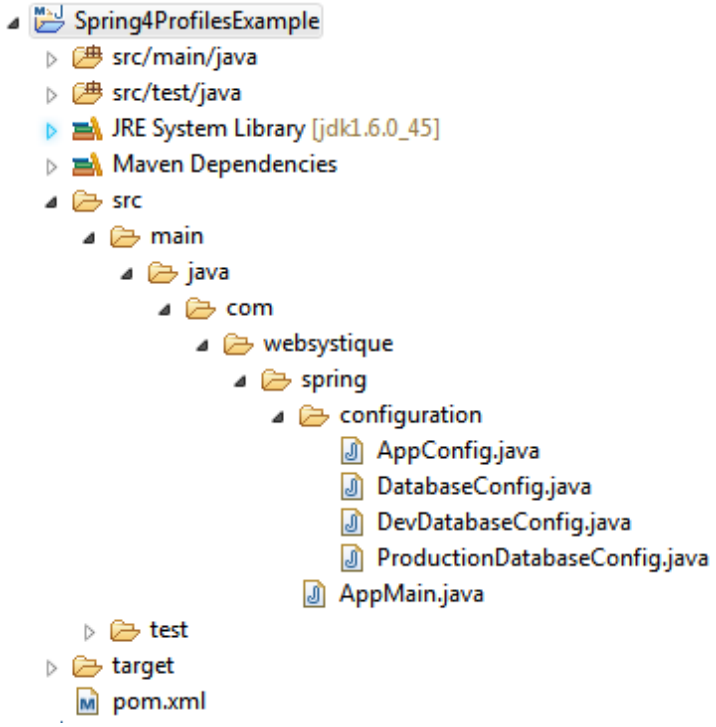
- Spring MVC 4 Form Validation and Resource Handling

Following technologies being used:

- Spring 4.0.6.RELEASE
- Maven 3
- JDK 1.6
- Eclipse JUNO Service Release 2

Project directory structure

Following will be the final project directory structure for this example:



Let's now add the content mentioned in above structure explaining each in detail.

Step 1: Provide Spring dependencies in Maven pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http:
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://ma
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.websystique.spring</groupId>
    <artifactId>Spring4ProfilesExample</artifactId>
    <version>1.0.0</version>
    <packaging>jar</packaging>
```

```
    <name>Spring4ProfilesExample</name>

    <properties>
        <springframework.version>4.0.6.RELEASE</springframework.ver
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>${springframework.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>${springframework.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-orm</artifactId>
            <version>${springframework.version}</version>
        </dependency>

    </dependencies>

</project>
```

Step 2: Create Spring Configuration Class

Spring configuration class are the ones annotated with `@Configuration`. These classes contains methods annotated with `@Bean`. These @Bean annotated methods generates beans managed by Spring container.

```
package com.websystique.spring.configuration;

import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan(basePackages = "com.websystique.spring")
public class AppConfig {

    @Autowired
    public DataSource dataSource;


}
```

Above Configuration just have one property to be auto-wired. Now what we will show here is that this dataSource bean can be injected with different beans on different environment (MySQL dataSource on Development & ORACLE for Production e.g.).

```
package com.websystique.spring.configuration;
```

```
import javax.sql.DataSource;

public interface DatabaseConfig {

    DataSource createDataSource();

}
```

Simple Interface to be implemented by all possible environment configurations.

```
package com.websystique.spring.configuration;

import javax.sql.DataSource;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.jdbc.datasource.DriverManagerDataSource;

@Profile("Development")
@Configuration
public class DevDatabaseConfig implements DatabaseConfig {

    @Override
    @Bean
    public DataSource createDataSource() {
        System.out.println("Creating DEV database");
        DriverManagerDataSource dataSource = new DriverManagerDataS
        /*
         * Set MySQL specific properties for Development Environmer
         */
        return dataSource;
    }

}
```

```
package com.websystique.spring.configuration;

import javax.sql.DataSource;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.jdbc.datasource.DriverManagerDataSource;

@Profile("Production")
@Configuration
public class ProductionDatabaseConfig implements DatabaseConfig {

    @Override
    @Bean
    public DataSource createDataSource() {
        System.out.println("Creating Production database");
        DriverManagerDataSource dataSource = new DriverManagerDataS
        /*
         * Set ORACLE specific properties for Production environmer
         */
        return dataSource;
    }

}
```

Both DevDatabaseConfig and ProductionDatabaseConfig are simple configuration classes implementing DatabaseConfig interface. What special about these classes are that they are annotated with `@Profile` annotation.

`@Profile` annotation on a component registers that component in Spring context only when that profile is active. Profile activation means this profile value should be available either by

- Setting `spring.profiles.active` property (via JVM arguments, environment variable or Servlet context parameter in web.xml in case of web applications)
- ApplicationContext.getEnvironment().setActiveProfiles("ProfileName");

Based on your environment you will provide the value of profile, and the beans dependent on that profile will be registered in Spring container.

Step 3: Create Main to run as Java Application

```
package com.websystique.spring;

import org.springframework.context.annotation.AnnotationConfigAppli

public class AppMain {

    public static void main(String args[]){
        AnnotationConfigApplicationContext  context = new Annotatic
        //Sets the active profiles
        context.getEnvironment().setActiveProfiles("Development");
        //Scans the mentioned package[s] and register all the @Comp
        context.scan("com.websystique.spring");
        context.refresh();
        context.close();
    }

}
```

Notice how we have set the active profiles to be used on runtime. `context.scan("package")` scans the mentioned package and registers all the @Component but when it encounters the @Profile annotation on a bean/configuration, it compare the profile value with the one supplied in environment. If the value mathces, it registers that bean else it skips it. In our case it's the DevDatabaseConfig dataDource which will be injected in AppConfig.dataSource.

Run above program , you will see following output:

```
Creating DEV database
```

That's it.

XML Based Spring Profile Configurtrion

Now, Let's discuss Spring Profile once more, this time using XML configuration

Replace DevelopmentDatabaseConfig by dev-config-context.xml (`src/main/resources/dev-config-context.xml`)

```
<?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/b


    <bean id="dataSource" class="org.springframework.jdbc.datasourc
        <property name="driverClassName" value="com.mysql.jdbc.Driv
        <property name="url" value="jdbc:mysql://localhost:3306/web
        <property name="username" value="myuser" />
        <property name="password" value="mypassword" />
    </bean>

</beans>
```

Replace ProductionDatabaseConfig by prod-config-context.xml (`src/main/resources/prod-config-context.xml`)

```
<?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

    <bean id="dataSource" class="org.springframework.jdbc.datasourc
        <property name="driverClassName" value=" oracle.jdbc.driver
        <property name="url"      value="jdbc:oracle:thin:@PRODHOST:
        <property name="username" value="myproduser" />
        <property name="password" value="myprodpassword" />
    </bean>

</beans>
```

Replace AppConfig by app-config.xml (`src/main/resources/app-config.xml`)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:context="http://www.springframework.org/schema/contex
        xsi:schemaLocation="http://www.springframework.org/schema/b
                            http://www.springframework.org/schema/c


    <context:component-scan base-package="com.websystique.spring"/>
```

```
    <beans profile="Development">
        <import resource="dev-config-context.xml"/>
    </beans>

    <beans profile="Production">
        <import resource="prod-config-context.xml"/>
    </beans>

</beans>
```

Based on actual profile in use, corresponding config-context.xml file will be loaded, other one will be ignored.

Let's also tweak the main:

```
package com.websystique.spring;

import org.springframework.context.support.AbstractApplicationConte
import org.springframework.context.support.ClassPathXmlApplicationC

public class AppMain {

    public static void main(String args[]){
        AbstractApplicationContext  context = new ClassPathXmlAppli
        //Sets the active profiles
        context.getEnvironment().setActiveProfiles("Development");
        /*
         * Perform any logic here
         */
        context.close();
    }

}
```

Run it. You will see the same output as with Annotation based setup.

Download Source Code

Download Now!

References

- Spring framework
- @Profile

me on Facebook , Google Plus & Twitter as well? I would love to hear your thoughts on these articles, it will help me improve further our learning process.

If you appreciate the effort I have put in this learning site, help me improve the visibility of this site towards global audience by sharing and linking this site from within and beyond your network. You & your friends can always link my site from your site on www.websystique.com, and share the learning.

After all, we are here to learn together, aren't we?

Related Posts:

1. **Spring Beans Auto-wiring Example using XML Configuration**
2. **Spring Dependency Injection Example with Constructor and Property Setter (XML)**
3. **Spring Dependency Injection Annotation Example, Beans Auto-wiring using @Autowired, @Qualifier & @Resource Annotations Configuration**
4. **Spring 4 + Hibernate 4 + MySQL+ Maven Integration example (Annotations+XML)**

spring. permalink.

← Spring @PropertySource & @Value annotations example

Spring 4 + Hibernate 4 + MySQL+ Maven Integration example (Annotations+XML) →