EBook Gratis

APRENDIZAJE spring

Free unaffiliated eBook created from
Stack Overflow contributors.

#spring

Tabla de contenido

Acerca de ... 1

Capítulo 1: Empezando con la primavera ... 2

Observaciones ... 2

Versiones ... 2

Examples ... 2

Configuración (Configuración XML) ... 2

Presentando las características principales de Spring Spring por ejemplo ... 3

Descripción ... 4

Dependencias ... 4

Clase principal ... 4

Archivo de configuración de la aplicación ... 5

Declaración de frijol por anotación ... 5

Declaración de frijol por configuración de la aplicación ... 6

Inyección Constructor ... 6

Inyección de propiedad ... 6

PostConstruct / PreDestroy hooks ... 7

¿Qué es Spring Framework, por qué deberíamos ir por él? ... 7

Entonces, ¿por qué elegir la primavera como puntales está ahí ... 8

Capítulo 2: Alcances de frijol ... 10

Examples ... 10

Alcance singleton ... 10

Habas singleton perezosos ... 11

Alcance prototipo ... 11

Ámbitos adicionales en contextos web-conscientes ... 12

Configuración XML ... 13

Configuración de Java (antes de la primavera 4.3) ... 13

Configuración de Java (después de la primavera 4.3) ... 14

Componentes impulsados por anotación ... 14

Capítulo 3: Configuración de ApplicationContext ... 16

Observaciones ... 16

Examples ... 16
Configuración de Java ... 16
Configuración xml ... 18
Auto cableado ... 19
Bootstrapping the ApplicationContext ... 20
Configuración de Java ... 20
Configuración xml ... 21
Auto cableado ... 21
Capítulo 4: Creando y usando frijoles ... 22
Examples ... 22
Autowiring todos los frijoles de un tipo específico ... 22
Declarando frijol ... 23
Anotación básica autowiring ... 24
Usando FactoryBean para la instanciación dinámica de frijol ... 25
Inyectar frijoles prototipo en singletons ... 26
Autowiring instancias de bean específicas con @Qualifier ... 29
Autowiring instancias específicas de clases usando parámetros de tipo genérico ... 30
Capítulo 5: Entendiendo el dispatcher-servlet.xml ... 33
Introducción ... 33
Examples ... 33
dispatcher-servlet.xml ... 33
Configuración del servlet del despachador en web.xml ... 34
Capítulo 6: Fuente de la propiedad ... 35
Examples ... 35
Anotación ... 35
Ejemplo de configuración xml usando PropertyPlaceholderConfigurer ... 35
Capítulo 7: Inicialización perezosa de primavera ... 37
Examples ... 37
Inicialización perezosa en la clase de configuración ... 37
Para escaneo de componentes y cableado automático ... 37
Ejemplo de Lazy Init en primavera ... 37
Capítulo 8: Inyección de dependencia (DI) e Inversión de control (IoC) ... 39

Observaciones ... 39
Examples ... 40
Inyectar una dependencia manualmente a través de la configuración XML ... 40
Inyectando una dependencia manualmente a través de la configuración de Java ... 41
Autowiring una dependencia a través de la configuración XML ... 42
Autowiring una dependencia a través de la configuración de Java ... 43
Capítulo 9: Lenguaje de Expresión de Primavera (SpEL) ... 45
Examples ... 45
Referencia de sintaxis ... 45
Expresiones literales ... 45
Lista en linea ... 45
Mapas en linea ... 45
Métodos de invocación ... 45
Capítulo 10: Núcleo de primavera ... 46
Examples ... 46
Introducción a Spring Core ... 46
¿Entendiendo cómo Spring gestiona la dependencia? ... 47
Capítulo 11: Obteniendo un SqlRowSet de SimpleJdbcCall ... 51
Introducción ... 51
Examples ... 51
Creación SimpleJdbcCall ... 51
Bases de datos Oracle ... 52
Capítulo 12: Perfil de primavera ... 54
Examples ... 54
Spring Profiles permite configurar partes disponibles para ciertos entornos ... 54
Capítulo 13: Plantilla de descanso ... 55
Examples ... 55
Descarga de un archivo grande ... 55
Uso de la autenticación básica preventiva con RestTemplate y HttpClient ... 55
Usando la autenticación básica con HttpComponent's HttpClient ... 57
Configuración de encabezados en Spring RestTemplate request ... 57
Resultados genéricos de Spring RestTemplate ... 58

Capítulo 14: Plantilla Jdbc 60
Introducción 60
Examples 60
Métodos básicos de consulta 60
Consulta de lista de mapas 60
SQLRowSet 61
Operaciones por lotes 61
NamedParameterJdbcTemplate extensión de JdbcTemplate 62
Capítulo 15: Registro condicional de frijol en primavera. 64
Observaciones 64
Examples 64
Registrar beans solo cuando se especifica una propiedad o valor 64
Anotaciones de condición 65
Condiciones de clase 65
Condiciones de frijol 65
Condiciones de propiedad 65
Condiciones de recursos 66
Condiciones de la aplicación web 66
Condiciones de expresión SpEL 66
Capítulo 16: SOAP WS DE CONSUMO 67
Examples 67
Consumiendo un SOAP WS con autenticación básica 67
Capítulo 17: Tarea de Ejecución y Programación 68
Examples 68
Habilitar la programación 68
Retraso fijo 68
Tipo de interés fijo 68
Cron expresión 68
Capítulo 18: Validación de frijol Spring JSR 303 72
Introducción 72
Examples 72

JSR303 Validación basada en anotaciones en ejemplos de resortes 72
Spring JSR 303 Validation - Personaliza los mensajes de error 74
Uso de @Valid para validar POJOs anidados 76
Creditos 78

Acerca de

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

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

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

Capítulo 1: Empezando con la primavera

Observaciones

Spring Framework es un marco de aplicación de código abierto e inversión del contenedor de control para la plataforma Java.

Versiones

Versión	Fecha de lanzamiento
4.3.x	2016-06-10
4.2.x	2015-07-31
4.1.x	2014-09-14
4.0.x	2013-12-12
3.2.x	2012-12-13
3.1.x	2011-12-13
3.0.x	2009-12-17
2.5.x	2007-12-25
2.0.x	2006-10-04
1.2.x	2005-05-13
1.1.x	2004-09-05
1.0.x	2003-03-24

Examples

Configuración (Configuración XML)

Pasos para crear Hello Spring:

0. Investigue Spring Boot para ver si eso se adapta mejor a sus necesidades.
1. Tener un proyecto configurado con las dependencias correctas. Se recomienda que esté utilizando Maven o Gradle .
2. crear una clase POJO, por ejemplo, `Employee.java`
3. cree un archivo XML donde pueda definir su clase y variables. ej. `beans.xml`

4. crea tu clase principal, por ejemplo, `Customer.java`
5. Incluya las judías de primavera (y sus dependencias transitivas) como una dependencia.

`Employee.java` :

```
package com.test;

public class Employee {

    private String name;

    public String getName() {
        return name;
    }

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

    public void displayName() {
        System.out.println(name);
    }
}
```

`beans.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
    http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">

    <bean id="employee" class="com.test.Employee">
        <property name="name" value="test spring"></property>
    </bean>

</beans>
```

`Customer.java` :

```
package com.test;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Customer {
   public static void main(String[] args) {
      ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");

      Employee obj = (Employee) context.getBean("employee");
      obj.displayName();
   }
}
```

Presentando las características principales de Spring Spring por ejemplo

Descripción

Este es un ejemplo de ejecución autónomo que incluye / muestra: *dependencias* mínimas necesarias, *configuración de* Java, *declaración de Bean* por anotación y configuración de Java, *inyección de dependencia* por el constructor y por la propiedad, y *enlaces Pre / Post* .

Dependencias

Estas dependencias son necesarias en el classpath:

1. núcleo de primavera
2. contexto primaveral
3. judías de primavera
4. primavera-aop
5. expresión de primavera
6. registro comunal

Clase principal

Comenzando desde el final, esta es nuestra clase principal que sirve como un marcador de posición para el método `main()` que inicializa el contexto de la aplicación al señalar la clase de configuración y carga todos los diversos beans necesarios para mostrar una funcionalidad particular.

```
package com.stackoverflow.documentation;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;



public class Main {

    public static void main(String[] args) {

        //initializing the Application Context once per application.
        ApplicationContext applicationContext =
                new AnnotationConfigApplicationContext(AppConfig.class);

        //bean registered by annotation
        BeanDeclaredByAnnotation beanDeclaredByAnnotation =
                applicationContext.getBean(BeanDeclaredByAnnotation.class);
        beanDeclaredByAnnotation.sayHello();

        //bean registered by Java configuration file
        BeanDeclaredInAppConfig beanDeclaredInAppConfig =
                applicationContext.getBean(BeanDeclaredInAppConfig.class);
        beanDeclaredInAppConfig.sayHello();

        //showcasing constructor injection
        BeanConstructorInjection beanConstructorInjection =
```

```
                applicationContext.getBean(BeanConstructorInjection.class);
        beanConstructorInjection.sayHello();

        //showcasing property injection
        BeanPropertyInjection beanPropertyInjection =
                applicationContext.getBean(BeanPropertyInjection.class);
        beanPropertyInjection.sayHello();

        //showcasing PreConstruct / PostDestroy hooks
        BeanPostConstructPreDestroy beanPostConstructPreDestroy =
                applicationContext.getBean(BeanPostConstructPreDestroy.class);
        beanPostConstructPreDestroy.sayHello();
    }
}
```

Archivo de configuración de la aplicación

La clase de configuración está anotada por `@Configuration` y se usa como un parámetro en el contexto de aplicación inicializado. La anotación `@ComponentScan` en el nivel de clase de la clase de configuración apunta a un paquete que se analizará en busca de Beans y dependencias registradas usando anotaciones. Finalmente, la anotación `@Bean` sirve como una definición de bean en la clase de configuración.

```
package com.stackoverflow.documentation;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan("com.stackoverflow.documentation")
public class AppConfig {

    @Bean
    public BeanDeclaredInAppConfig beanDeclaredInAppConfig() {
        return new BeanDeclaredInAppConfig();
    }
}
```

Declaración de frijol por anotación

La anotación `@Component` sirve para demarcar el POJO como un bean Spring disponible para el registro durante el escaneo de componentes.

```
@Component
public class BeanDeclaredByAnnotation {

    public void sayHello() {
        System.out.println("Hello, World from BeanDeclaredByAnnotation !");
    }
}
```

Declaración de frijol por configuración de la aplicación

Tenga en cuenta que no necesitamos anotar o marcar nuestro POJO, ya que la declaración / definición del bean está ocurriendo en el archivo de clase de configuración de la aplicación.

```
public class BeanDeclaredInAppConfig {

    public void sayHello() {
        System.out.println("Hello, World from BeanDeclaredInAppConfig !");
    }
}
```

Inyección Constructor

Observe que la anotación `@Autowired` se establece en el nivel de constructor. También tenga en cuenta que, a menos que se defina explícitamente por nombre, el cableado automático predeterminado se realiza *según el tipo* de bean (en este caso, `BeanToBeInjected`).

```
package com.stackoverflow.documentation;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class BeanConstructorInjection {

    private BeanToBeInjected dependency;

    @Autowired
    public BeanConstructorInjection(BeanToBeInjected dependency) {
        this.dependency = dependency;
    }

    public void sayHello() {
        System.out.print("Hello, World from BeanConstructorInjection with dependency: ");
        dependency.sayHello();
    }
}
```

Inyección de propiedad

Observe que la anotación `@Autowired` delimita el método de establecimiento cuyo nombre sigue el estándar JavaBeans.

```
package com.stackoverflow.documentation;


import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
```

```
@Component
public class BeanPropertyInjection {

    private BeanToBeInjected dependency;

    @Autowired
    public void setBeanToBeInjected(BeanToBeInjected beanToBeInjected) {
        this.dependency = beanToBeInjected;
    }

    public void sayHello() {
        System.out.println("Hello, World from BeanPropertyInjection !");
    }
}
```

PostConstruct / PreDestroy hooks

Podemos interceptar la inicialización y la destrucción de un Bean mediante los ganchos `@PostConstruct` y `@PreDestroy` .

```
package com.stackoverflow.documentation;

import org.springframework.stereotype.Component;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

@Component
public class BeanPostConstructPreDestroy {

    @PostConstruct
    public void pre() {
        System.out.println("BeanPostConstructPreDestroy - PostConstruct");
    }

    public void sayHello() {
        System.out.println(" Hello World, BeanPostConstructPreDestroy !");
    }

    @PreDestroy
    public void post() {
        System.out.println("BeanPostConstructPreDestroy - PreDestroy");
    }
}
```

¿Qué es Spring Framework, por qué deberíamos ir por él?

Spring es un marco que proporciona muchas clases. Al usar esto, no necesitamos escribir la lógica de la placa de la caldera en nuestro código, por lo que Spring proporciona una capa abstracta en J2ee.

Por ejemplo, en el programador de aplicaciones JDBC simple es responsable de

1. Cargando la clase de conductor
2. Creando la conexion
3. Creación de objeto de declaración
4. Manejando las excepciones.
5. Creando consulta
6. Ejecutando consulta
7. Cerrando la conexion

Que se trata como código repetitivo, ya que cada programador escribe el mismo código. Por lo tanto, para simplificar, el marco se ocupa de la lógica de repetición y el programador debe escribir solo lógica de negocios. Entonces, al usar Spring Framework, podemos desarrollar proyectos rápidamente con un mínimo de líneas de código, sin ningún error, el tiempo y el costo de desarrollo también se reducen.

Entonces, ¿por qué elegir la primavera como puntales está ahí

Strut es un marco que proporciona soluciones solo para aspectos web y struts es de naturaleza invasiva. La primavera tiene muchas características sobre los puntales, así que tenemos que elegir la primavera.

1. Spring ***no*** es de naturaleza ***invasiva*** : eso significa que no necesita extender ninguna clase ni implementar ninguna interfaz a su clase.
2. Spring es ***versátil*** : eso significa que puede integrarse con cualquier tecnología existente en su proyecto.
3. Spring proporciona desarrollo de proyectos de ***extremo a extremo*** : eso significa que podemos desarrollar todos los módulos como la capa empresarial, la capa de persistencia.
4. La primavera es ***liviana*** : eso significa que si desea trabajar en un módulo en particular, no necesita aprender la primavera completa, solo aprenda ese módulo en particular (por ejemplo, Spring Jdbc, Spring DAO)
5. La primavera soporta ***inyección de dependencia*** .
6. Spring admite el ***desarrollo de múltiples proyectos,*** por ejemplo: aplicación Core java, aplicación web, aplicación distribuida, aplicación empresarial.
7. Spring es compatible con la Programación orientada a Aspectos para asuntos transversales.

Así que finalmente podemos decir que la primavera es una alternativa a Struts. Pero Spring no es un reemplazo de la API de J2EE, ya que Spring proporcionó clases internamente a las clases de la API de J2EE. Spring es un marco amplio por lo que se ha dividido en varios módulos. Ningún módulo es dependiente de otro, excepto Spring Core. Algunos módulos importantes son

1. Núcleo de primavera
2. Primavera JDBC
3. Primavera AOP
4. Transacción de primavera
5. ORM de primavera
6. Primavera MVC

Lea Empezando con la primavera en línea: https://riptutorial.com/es/spring/topic/786/empezando-con-la-primavera

Capítulo 2: Alcances de frijol

Examples

Alcance singleton

Si un bean se define con el alcance de singleton, solo habrá una única instancia de objeto inicializada en el contenedor Spring. Todas las solicitudes a este bean devolverán la misma instancia compartida. Este es el alcance por defecto cuando se define un bean.

Dada la siguiente clase de MyBean:

```
public class MyBean {
    private static final Logger LOGGER = LoggerFactory.getLogger(MyBean.class);
    private String property;

    public MyBean(String property) {
        this.property = property;
        LOGGER.info("Initializing {} bean...", property);
    }

    public String getProperty() {
        return this.property;
    }

    public void setProperty(String property) {
        this.property = property;
    }
}
```

Podemos definir un bean singleton con la anotación @Bean:

```
@Configuration
public class SingletonConfiguration {

    @Bean
    public MyBean singletonBean() {
        return new MyBean("singleton");
    }
}
```

El siguiente ejemplo recupera el mismo bean dos veces del contexto de Spring:

```
MyBean singletonBean1 = context.getBean("singletonBean", MyBean.class);
singletonBean1.setProperty("changed property");

MyBean singletonBean2 = context.getBean("singletonBean", MyBean.class);
```

Al registrar la propiedad singletonBean2, se mostrará el mensaje *"propiedad modificada"* , ya que acabamos de recuperar la misma instancia compartida.

Dado que la instancia se comparte entre diferentes componentes, se recomienda definir el alcance de singleton para objetos sin estado.

Habas singleton perezosos

Por defecto, los beans singleton están pre-instanciados. Por lo tanto, la instancia de objeto compartido se creará cuando se cree el contenedor Spring. Si iniciamos la aplicación, se mostrará el mensaje *"Inicializando singleton bean ..."* .

Si no queremos que el bean sea pre-instanciado, podemos agregar la anotación @Lazy a la definición del bean. Esto evitará que el bean se cree hasta que se solicite por primera vez.

```
@Bean
@Lazy
public MyBean lazySingletonBean() {
    return new MyBean("lazy singleton");
}
```

Ahora, si iniciamos el contenedor Spring, no aparecerá el mensaje *"Inicializando el bean singleton lento ..."* . El bean no se creará hasta que se solicite por primera vez:

```
logger.info("Retrieving lazy singleton bean...");
context.getBean("lazySingletonBean");
```

Si ejecutamos la aplicación con los frijoles singleton y perezosos singleton definidos, producirá los siguientes mensajes:

```
Initializing singleton bean...
Retrieving lazy singleton bean...
Initializing lazy singleton bean...
```

Alcance prototipo

Un bean de ámbito prototipo no se crea previamente en el inicio del contenedor Spring. En su lugar, se creará una nueva instancia nueva cada vez que se envíe una solicitud para recuperar este bean al contenedor. Este alcance se recomienda para objetos con estado, ya que su estado no será compartido por otros componentes.

Para definir un bean de ámbito de prototipo, debemos agregar la anotación @Scope, especificando el tipo de ámbito que queremos.

Dada la siguiente clase de MyBean:

```
public class MyBean {
    private static final Logger LOGGER = LoggerFactory.getLogger(MyBean.class);
    private String property;

    public MyBean(String property) {
        this.property = property;
        LOGGER.info("Initializing {} bean...", property);
```

```
    }

    public String getProperty() {
        return this.property;
    }

    public void setProperty(String property) {
        this.property = property;
    }
}
```

Definimos una definición de bean, indicando su alcance como prototipo:

```
@Configuration
public class PrototypeConfiguration {

    @Bean
    @Scope("prototype")
    public MyBean prototypeBean() {
        return new MyBean("prototype");
    }
}
```

Para ver cómo funciona, recuperamos el bean del contenedor Spring y establecemos un valor diferente para su campo de propiedad. A continuación, volveremos a recuperar el bean del contenedor y buscaremos su valor:

```
MyBean prototypeBean1 = context.getBean("prototypeBean", MyBean.class);
prototypeBean1.setProperty("changed property");

MyBean prototypeBean2 = context.getBean("prototypeBean", MyBean.class);

logger.info("Prototype bean 1 property: " + prototypeBean1.getProperty());
logger.info("Prototype bean 2 property: " + prototypeBean2.getProperty());
```

Si observamos el siguiente resultado, podemos ver cómo se ha creado una nueva instancia en cada solicitud de bean:

```
Initializing prototype bean...
Initializing prototype bean...
Prototype bean 1 property: changed property
Prototype bean 2 property: prototype
```

Un error común es asumir que el bean se vuelve a crear por invocación o por hilo, este **NO** es el caso. En su lugar, se crea una instancia POR INYECCIÓN (o recuperación del contexto). Si un bean Prototipo con ámbito solo se inyecta en un único bean singleton, solo habrá una instancia de ese frijol Prototype.

Spring no gestiona el ciclo de vida completo de un bean prototipo: el contenedor crea instancias, configura, decora y ensambla un objeto prototipo, se lo entrega al cliente y luego no tiene conocimiento de esa instancia de prototipo.

Ámbitos adicionales en contextos web-conscientes

Hay varios ámbitos que están disponibles solo en un contexto de aplicación compatible con la web:

- **solicitud** - se crea una nueva instancia de bean por solicitud HTTP
- **sesión** - se crea una nueva instancia de bean por sesión HTTP
- **aplicación** - se crea una nueva instancia de bean por `ServletContext`
- **globalSession** : la nueva instancia de bean se crea por sesión global en el entorno Portlet (en el ámbito de sesión global del entorno Servlet es igual al alcance de la sesión)
- **websocket** - se crea una nueva instancia de bean por sesión de WebSocket

No se requiere configuración adicional para declarar y acceder a beans de ámbito web en el entorno Spring Web MVC.

Configuración XML

```
<bean id="myRequestBean" class="OneClass" scope="request"/>
<bean id="mySessionBean" class="AnotherClass" scope="session"/>
<bean id="myApplicationBean" class="YetAnotherClass" scope="application"/>
<bean id="myGlobalSessionBean" class="OneMoreClass" scope="globalSession"/>
```

Configuración de Java (antes de la primavera 4.3)

```
@Configuration
public class MyConfiguration {

    @Bean
    @Scope(value = WebApplicationContext.SCOPE_REQUEST, proxyMode =
ScopedProxyMode.TARGET_CLASS)
    public OneClass myRequestBean() {
        return new OneClass();
    }

    @Bean
    @Scope(value = WebApplicationContext.SCOPE_SESSION, proxyMode =
ScopedProxyMode.TARGET_CLASS)
    public AnotherClass mySessionBean() {
        return new AnotherClass();
    }

    @Bean
    @Scope(value = WebApplicationContext.SCOPE_APPLICATION, proxyMode =
ScopedProxyMode.TARGET_CLASS)
    public YetAnotherClass myApplicationBean() {
        return new YetAnotherClass();
    }

    @Bean
    @Scope(value = WebApplicationContext.SCOPE_GLOBAL_SESSION, proxyMode =
ScopedProxyMode.TARGET_CLASS)
    public OneMoreClass myGlobalSessionBean() {
        return new OneMoreClass();
    }
}
```

Configuración de Java (después de la primavera 4.3)

```
@Configuration
public class MyConfiguration {

    @Bean
    @RequestScope
    public OneClass myRequestBean() {
        return new OneClass();
    }

    @Bean
    @SessionScope
    public AnotherClass mySessionBean() {
        return new AnotherClass();
    }

    @Bean
    @ApplicationScope
    public YetAnotherClass myApplicationBean() {
        return new YetAnotherClass();
    }
}
```

Componentes impulsados por anotación

```
@Component
@RequestScope
public class OneClass {
    ...
}

@Component
@SessionScope
public class AnotherClass {
    ...
}

@Component
@ApplicationScope
public class YetAnotherClass {
    ...
}

@Component
@Scope(scopeName = WebApplicationContext.SCOPE_GLOBAL_SESSION, proxyMode =
ScopedProxyMode.TARGET_CLASS)
public class OneMoreClass {
    ...
}

@Component
@Scope(scopeName = "websocket", proxyMode = ScopedProxyMode.TARGET_CLASS)
public class AndOneMoreClass {
    ...
}
```

Lea Alcances de frijol en línea: https://riptutorial.com/es/spring/topic/3526/alcances-de-frijol

Capítulo 3: Configuración de ApplicationContext

Observaciones

Spring lo ha hecho para que la configuración de un `ApplicationContext` sea extremadamente flexible. Existen numerosas formas de aplicar cada tipo de configuración, y todas pueden combinarse y combinarse muy bien.

La configuración de Java es una forma de configuración *explícita* . Se `@Configuration` clase anotada `@Configuration` para especificar los beans que formarán parte de `ApplicationContext` , así como para definir y conectar las dependencias de cada bean.

La configuración xml es una forma de configuración *explícita* . Se utiliza un esquema xml específico para definir los beans que formarán parte de `ApplicationContext` . Este mismo esquema se utiliza para definir y conectar las dependencias de cada bean.

Autowiring es una forma de configuración *automática* . Ciertas anotaciones se utilizan en las definiciones de clase para establecer qué beans formarán parte de `ApplicationContext` , y otras anotaciones se utilizan para conectar las dependencias de estos beans.

Examples

Configuración de Java

La configuración de Java se realiza normalmente aplicando la anotación `@Configuration` a una clase para sugerir que una clase contiene definiciones de bean. Las definiciones de bean se especifican aplicando la anotación `@Bean` a un método que devuelve un objeto.

```
@Configuration // This annotation tells the ApplicationContext that this class
               // contains bean definitions.
class AppConfig {
    /**
     * An Author created with the default constructor
     * setting no properties
     */
    @Bean // This annotation marks a method that defines a bean
    Author author1() {
        return new Author();
    }

    /**
     * An Author created with the constructor that initializes the
     * name fields
     */
    @Bean
    Author author2() {
        return new Author("Steven", "King");
```

```
    }

    /**
     * An Author created with the default constructor, but
     * then uses the property setters to specify name fields
     */
    @Bean
    Author author3() {
        Author author = new Author();
        author.setFirstName("George");
        author.setLastName("Martin");
        return author;
    }

    /**
     * A Book created referring to author2 (created above) via
     * a constructor argument.  The dependency is fulfilled by
     * invoking the method as plain Java.
     */
    @Bean
    Book book1() {
        return new Book(author2(), "It");
    }

    /**
     * A Book created referring to author3 (created above) via
     * a property setter.  The dependency is fulfilled by
     * invoking the method as plain Java.
     */
    @Bean
    Book book2() {
        Book book = new Book();
        book.setAuthor(author3());
        book.setTitle("A Game of Thrones");
        return book;
    }
}
```

```
// The classes that are being initialized and wired above...
class Book { // assume package org.springframework.example
    Author author;
    String title;

    Book() {} // default constructor
    Book(Author author, String title) {
        this.author = author;
        this.title= title;
    }

    Author getAuthor() { return author; }
    String getTitle() { return title; }

    void setAuthor(Author author) {
        this.author = author;
    }

    void setTitle(String title) {
        this.title= title;
    }
}
```

```
class Author { // assume package org.springframework.example
    String firstName;
    String lastName;

    Author() {} // default constructor
    Author(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    String getFirstName() { return firstName; }
    String getLastName() { return lastName; }

    void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

Configuración xml

La configuración de XML se realiza normalmente mediante la definición de beans dentro de un archivo xml, utilizando el esquema de `beans` específico de Spring. Bajo el elemento de root `beans` , la definición típica de bean se haría usando el subelemento de `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.xsd">
    <!-- An Author created with the default constructor
         setting no properties -->
    <bean id="author1" class="org.springframework.example.Author" />

    <!-- An Author created with the constructor that initializes the
         name fields -->
    <bean id="author2" class="org.springframework.example.Author">
        <constructor-arg index="0" value="Steven" />
        <constructor-arg index="1" value="King" />
    </bean>

    <!-- An Author created with the default constructor, but
         then uses the property setters to specify name fields -->
    <bean id="author3" class="org.springframework.example.Author">
        <property name="firstName" value="George" />
        <property name="lastName" value="Martin" />
    </bean>

    <!-- A Book created referring to author2 (created above) via
         a constructor argument -->
    <bean id="book1" class="org.springframework.example.Book">
        <constructor-arg index="0" ref="author2" />
        <constructor-arg index="1" value="It" />
    </bean>
```

```
    <!-- A Book created referring to author3 (created above) via
         a property setter -->
    <bean id="book1" class="org.springframework.example.Book">
        <property name="author" ref="author3" />
        <property name="title" value="A Game of Thrones" />
    </bean>
</beans>
```

```
// The classes that are being initialized and wired above...
class Book { // assume package org.springframework.example
    Author author;
    String title;

    Book() {} // default constructor
    Book(Author author, String title) {
        this.author = author;
        this.title= title;
    }

    Author getAuthor() { return author; }
    String getTitle() { return title; }

    void setAuthor(Author author) {
        this.author = author;
    }

    void setTitle(String title) {
        this.title= title;
    }
}

class Author { // assume package org.springframework.example
    String firstName;
    String lastName;

    Author() {} // default constructor
    Author(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    String getFirstName() { return firstName; }
    String getLastName() { return lastName; }

    void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

Auto cableado

El cableado automático se realiza mediante una anotación de *esterotipo* para especificar qué clases serán beans en `ApplicationContext` y utilizando las anotaciones `Autowired` y `Value` para

especificar las dependencias de bean. La parte única de autowiring es que no hay una definición externa de `ApplicationContext` , ya que todo se realiza dentro de las clases que son los beans en sí.

```
@Component // The annotation that specifies to include this as a bean
           // in the ApplicationContext
class Book {

    @Autowired // The annotation that wires the below defined Author
               // instance into this bean
    Author author;

    String title = "It";

    Author getAuthor() { return author; }
    String getTitle() { return title; }
}

@Component // The annotation that specifies to include
           // this as a bean in the ApplicationContext
class Author {
    String firstName = "Steven";
    String lastName = "King";

    String getFirstName() { return firstName; }
    String getLastName() { return lastName; }
}
```

Bootstrapping the ApplicationContext

Configuración de Java

La clase de configuración solo necesita ser una clase que se encuentre en la ruta de clase de su aplicación y sea visible para la clase principal de su aplicación.

```
class MyApp {
    public static void main(String[] args) throws Exception {
        AnnotationConfigApplicationContext appContext =
            new AnnotationConfigApplicationContext(MyConfig.class);

        // ready to retrieve beans from appContext, such as myObject.
    }
}

@Configuration
class MyConfig {
    @Bean
    MyObject myObject() {
        // ...configure myObject...
    }

    // ...define more beans...
}
```

Configuración xml

El archivo xml de configuración solo necesita estar en la ruta de clase de su aplicación.

```
class MyApp {
    public static void main(String[] args) throws Exception {
        ClassPathXmlApplicationContext appContext =
            new ClassPathXmlApplicationContext("applicationContext.xml");

        // ready to retrieve beans from appContext, such as myObject.
    }
}

<?xml version="1.0" encoding="UTF-8"?>
<!-- applicationContext.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.xsd">
    <bean id="myObject" class="com.example.MyObject">
        <!-- ...configure myObject... -->
    </bean>

    <!-- ...define more beans... -->
</beans>
```

Auto cableado

El cableado automático necesita saber qué paquetes base analizar para beans anotados (`@Component`). Esto se especifica mediante el `#scan(String...)` .

```
class MyApp {
    public static void main(String[] args) throws Exception {
        AnnotationConfigApplicationContext appContext =
            new AnnotationConfigApplicationContext();
        appContext.scan("com.example");
        appContext.refresh();

        // ready to retrieve beans from appContext, such as myObject.
    }
}

// assume this class is in the com.example package.
@Component
class MyObject {
    // ...myObject definition...
}
```

Lea Configuración de ApplicationContext en línea:
https://riptutorial.com/es/spring/topic/3844/configuracion-de-applicationcontext

Capítulo 4: Creando y usando frijoles

Examples

Autowiring todos los frijoles de un tipo específico

Si tiene varias implementaciones de la misma interfaz, Spring puede convertirlas en un objeto de colección. Voy a usar un ejemplo usando un patrón Validator [1]

Foo Class:

```
public class Foo {
     private String name;
     private String emailAddress;
     private String errorMessage;
     /** Getters & Setters omitted **/
}
```

Interfaz:

```
public interface FooValidator {
    public Foo validate(Foo foo);
}
```

Nombre Validador Clase:

```
@Component(value="FooNameValidator")
public class FooNameValidator implements FooValidator {
    @Override
    public Foo validate(Foo foo) {
        //Validation logic goes here.
    }
}
```

Correo electrónico Validador de clase:

```
@Component(value="FooEmailValidator")
public class FooEmailValidator implements FooValidator {
    @Override
    public Foo validate(Foo foo) {
        //Different validation logic goes here.
    }
}
```

Ahora puede habilitar automáticamente estos validadores individualmente o juntos en una clase.

Interfaz:

```
public interface FooService {
    public void handleFoo(Foo foo);
```

```
}
```

Clase:

```
@Service
public class FooServiceImpl implements FooService {
    /** Autowire all classes implementing FooValidator interface**/
    @Autowired
    private List<FooValidator> allValidators;

    @Override
    public void handleFoo(Foo foo) {
       /**You can use all instances from the list**/
       for(FooValidator validator : allValidators) {
           foo = validator.validate(foo);
       }
    }
}
```

Vale la pena señalar que si tiene más de una implementación de una interfaz en el contenedor Spring IoC y no especifica cuál desea usar con la anotación `@Qualifier` , Spring lanzará una excepción cuando intente iniciar, porque ganó " No sé qué instancia utilizar.

1: Esta no es la forma correcta de hacer validaciones tan simples. Este es un ejemplo simple sobre autowiring. Si desea tener una idea de un método de validación mucho más sencillo, vea cómo Spring realiza la validación con anotaciones.

Declarando frijol

Para declarar un bean, simplemente anote un método con la anotación `@Bean` o anote una clase con la anotación `@Component` (las anotaciones `@Service` , `@Repository` , `@Controller` podrían usarse también).

Cuando JavaConfig encuentra tal método, ejecutará ese método y registrará el valor de retorno como un bean dentro de un BeanFactory. Por defecto, el nombre del bean será el del nombre del método.

Podemos crear frijol usando una de tres maneras:

1. **Usando la configuración basada en Java** : En el archivo de configuración necesitamos declarar el bean usando la anotación @bean

   ```
   @Configuration
   public class AppConfig {
       @Bean
       public TransferService transferService() {
           return new TransferServiceImpl();
       }
   }
   ```

2. **Uso de la configuración basada en XML** : para la configuración basada en XML necesitamos crear el bean declarar en la configuración de la aplicación XML, es decir

```
<beans>
    <bean name="transferService" class="com.acme.TransferServiceImpl"/>
</beans>
```

3. **Componente** controlado por anotación: para los componentes controlados por anotación, debemos agregar la anotación @Component a la clase que queremos declarar como bean.

```
@Component("transferService")
public class TransferServiceImpl implements TransferService {
    ...
}
```

Ahora los tres beans con nombre `transferService` están disponibles en `BeanFactory` o `ApplicationContext` .

Anotación básica autowiring

Interfaz:

```
public interface FooService {
    public int doSomething();
}
```

Clase:

```
@Service
public class FooServiceImpl implements FooService {
    @Override
    public int doSomething() {
        //Do some stuff here
        return 0;
    }
}
```

Se debe tener en cuenta que una clase debe implementar una interfaz para que Spring pueda conectar automáticamente esta clase. Existe un método para permitir que Spring autowire clases autónomas utilizando el tiempo de carga de tejido, pero está fuera del alcance de este ejemplo.

Puede obtener acceso a este bean en cualquier clase que haya sido instanciada por el contenedor Spring IoC utilizando la anotación `@Autowired` .

Uso:

```
@Autowired([required=true])
```

La anotación `@Autowired` intentará primero autowire por tipo y luego volverá al nombre del bean en caso de ambigüedad.

Esta anotación se puede aplicar de varias maneras diferentes.

Inyección de constructor:

```
public class BarClass() {
    private FooService fooService

    @Autowired
    public BarClass(FooService fooService) {
        this.fooService = fooService;
    }
}
```

Inyección de campo:

```
public class BarClass() {
    @Autowired
    private FooService fooService;
}
```

Inyección de Setter:

```
public class BarClass() {
    private FooService fooService;

    @Autowired
    public void setFooService(FooService fooService) {
        this.fooService = fooService;
    }
}
```

Usando FactoryBean para la instanciación dinámica de frijol

Para decidir dinámicamente qué frijoles inyectar, podemos usar `FactoryBean` s. Estas son clases que implementan el patrón de método de fábrica, proporcionando instancias de beans para el contenedor. Son reconocidos por Spring y se pueden usar de manera transparente, sin necesidad de saber que el frijol proviene de una fábrica. Por ejemplo:

```
public class ExampleFactoryBean extends AbstractFactoryBean<String> {
    // This method determines the type of the bean for autowiring purposes
    @Override
    public Class<?> getObjectType() {
        return String.class;
    }

    // this factory method produces the actual bean
    @Override
    protected String createInstance() throws Exception {
        // The thing you return can be defined dynamically,
        // that is read from a file, database, network or just
        // simply randomly generated if you wish.
        return "Something from factory";
    }
}
```

Configuración:

```
@Configuration
```

```
public class ExampleConfig {
    @Bean
    public FactoryBean<String> fromFactory() {
        return new ExampleFactoryBean();
    }
}
```

Obteniendo el frijol

```
AbstractApplicationContext context = new
AnnotationConfigApplicationContext(ExampleConfig.class);
String exampleString = (String) context.getBean("fromFactory");
```

Para obtener el FactoryBean real, use el prefijo ampersand antes del nombre del bean:

```
FactoryBean<String> bean = (FactoryBean<String>) context.getBean("&fromFactory");
```

Tenga en cuenta que solo puede usar `prototype` o ámbitos `singleton` : para cambiar el alcance al método de anulación de `prototype` anulación `isSingleton` :

```
public class ExampleFactoryBean extends AbstractFactoryBean<String> {
    @Override
    public boolean isSingleton() {
        return false;
    }

    // other methods omitted for readability reasons
}
```

Tenga en cuenta que el alcance se refiere a las instancias reales que se crean, no al propio bean de fábrica.

Inyectar frijoles prototipo en singletons

El contenedor crea un bean singleton e inyecta colaboradores en él solo una vez. Este no es el comportamiento deseado cuando un bean singleton tiene un colaborador de ámbito prototipo, ya que el bean de ámbito prototipo debe inyectarse cada vez que se accede a él mediante el acceso.

Hay varias soluciones a este problema:

1. Utilice el método de búsqueda de inyección
2. Recupere un bean de ámbito de prototipo a través de `javax.inject.Provider`
3. Recupere un bean de ámbito prototipo a través de `org.springframework.beans.factory.ObjectFactory` (un equivalente de # 2, pero con la clase que es específica de Spring)
4. Haga que un contenedor de bean singleton sea consciente mediante la implementación de la interfaz `ApplicationContextAware`

Los enfoques # 3 y # 4 generalmente no se recomiendan, ya que vinculan fuertemente una aplicación al marco de Spring. Por lo tanto, no están cubiertos en este ejemplo.

Inyección del método de búsqueda a través de la configuración XML y un método abstracto.

Clases de Java

```
public class Window {
}

public abstract class WindowGenerator {

    public Window generateWindow() {
        Window window = createNewWindow(); // new instance for each call
        ...
    }

    protected abstract Window createNewWindow(); // lookup method
}
```

XML

```
<bean id="window" class="somepackage.Window" scope="prototype" lazy-init="true"/>

<bean id="windowGenerator" class="somepackage.WindowGenerator">
    <lookup-method name="createNewWindow" bean="window"/>
</bean>
```

Método de búsqueda de inyección a través de la configuración de Java y @Component.

Clases de Java

```
public class Window {
}

@Component
public class WindowGenerator {

    public Window generateWindow() {
        Window window = createNewWindow(); // new instance for each call
        ...
    }

    @Lookup
    protected Window createNewWindow() {
        throw new UnsupportedOperationException();
    }
}
```

Configuración de Java

```
@Configuration
@ComponentScan("somepackage") // package where WindowGenerator is located
public class MyConfiguration {

    @Bean
    @Lazy
    @Scope(scopeName = ConfigurableBeanFactory.SCOPE_PROTOTYPE)
```

```
    public Window window() {
        return new Window();
    }
}
```

Inyección del método de búsqueda manual a través de la configuración de Java.

Clases de Java

```
public class Window {
}

public abstract class WindowGenerator {

    public Window generateWindow() {
        Window window = createNewWindow(); // new instance for each call
        ...
    }

    protected abstract Window createNewWindow(); // lookup method
}
```

Configuración de Java

```
@Configuration
public class MyConfiguration {

    @Bean
    @Lazy
    @Scope(scopeName = ConfigurableBeanFactory.SCOPE_PROTOTYPE)
    public Window window() {
        return new Window();
    }

    @Bean
    public WindowGenerator windowGenerator(){
        return new WindowGenerator() {
            @Override
            protected Window createNewWindow(){
                return window();
            }
        };
    }
}
```

Inyección de un bean de ámbito protoype en singleton a través de `javax.inject.Provider`

Clases de java

```
public class Window {
}

public class WindowGenerator {

    private final Provider<Window> windowProvider;

    public WindowGenerator(final Provider<Window> windowProvider) {
```

```
        this.windowProvider = windowProvider;
    }

    public Window generateWindow() {
        Window window = windowProvider.get(); // new instance for each call
        ...
    }
}
```

XML

```
<bean id="window" class="somepackage.Window" scope="prototype" lazy-init="true"/>

<bean id="windowGenerator" class="somepackage.WindowGenerator">
    <constructor-arg>
        <bean class="org.springframework.beans.factory.config.ProviderCreatingFactoryBean">
            <property name="targetBeanName" value="window"/>
        </bean>
    </constructor-arg>
</bean>
```

Los mismos enfoques también se pueden utilizar para otros ámbitos (por ejemplo, para inyectar un bean de ámbito de solicitud en singleton).

Autowiring instancias de bean específicas con @Qualifier

Si tiene varias implementaciones de la misma interfaz, Spring necesita saber cuál de ellas debería incluir en una clase. Voy a usar un patrón Validator en este ejemplo. [1]

Foo Class:

```
public class Foo {
     private String name;
     private String emailAddress;
     private String errorMessage;
     /** Getters & Setters omitted **/
}
```

Interfaz:

```
public interface FooValidator {
    public Foo validate(Foo foo);
}
```

Nombre Validador Clase:

```
@Component(value="FooNameValidator")
public class FooNameValidator implements FooValidator {
    @Override
    public Foo validate(Foo foo) {
        //Validation logic goes here.
    }
}
```

Correo electrónico Validador de clase:

```
@Component(value="FooEmailValidator")
public class FooEmailValidator implements FooValidator {
    @Override
    public Foo validate(Foo foo) {
        //Different validation logic goes here.
    }
}
```

Ahora puede autowire estos validadores individualmente en una clase.

Interfaz:

```
public interface FooService {
    public void handleFoo(Foo foo);
}
```

Clase:

```
@Service
public class FooServiceImpl implements FooService {
    /** Autowire validators individually **/
    @Autowired
    /*
     * Notice how the String value here matches the value
     * on the @Component annotation? That's how Spring knows which
     * instance to autowire.
     */
    @Qualifier("FooNameValidator")
    private FooValidator nameValidator;

    @Autowired
    @Qualifier("FooEmailValidator")
    private FooValidator emailValidator;

    @Override
    public void handleFoo(Foo foo) {
        /**You can use just one instance if you need**/
        foo = nameValidator.validate(foo);
    }
}
```

Vale la pena señalar que si tiene más de una implementación de una interfaz en el contenedor Spring IoC y no especifica cuál desea usar con la anotación `@Qualifier` , Spring lanzará una excepción cuando intente iniciar, porque ganó " No sé qué instancia utilizar.

1: Esta no es la forma correcta de hacer validaciones tan simples. Este es un ejemplo simple sobre autowiring. Si desea tener una idea de un método de validación mucho más sencillo, vea cómo Spring realiza la validación con anotaciones.

Autowiring instancias específicas de clases usando parámetros de tipo genérico

Si tiene una interfaz con un parámetro de tipo genérico, Spring puede usarlo solo para implementaciones de autowire que implementan un parámetro de tipo que usted especifique.

Interfaz:

```
public interface GenericValidator<T> {
    public T validate(T object);
}
```

Clase Validador Foo:

```
@Component
public class FooValidator implements GenericValidator<Foo> {
    @Override
    public Foo validate(Foo foo) {
        //Logic here to validate foo objects.
    }
}
```

Clase Validador de Barras:

```
@Component
public class BarValidator implements GenericValidator<Bar> {
    @Override
    public Bar validate(Bar bar) {
        //Bar validation logic here
    }
}
```

Ahora puede activar automáticamente estos validadores utilizando parámetros de tipo para decidir qué instancia autowire.

Interfaz:

```
public interface FooService {
    public void handleFoo(Foo foo);
}
```

Clase:

```
@Service
public class FooServiceImpl implements FooService {
    /** Autowire Foo Validator **/
    @Autowired
    private GenericValidator<Foo> fooValidator;

    @Override
    public void handleFoo(Foo foo) {
        foo = fooValidator.validate(foo);
    }
}
```

Lea Creando y usando frijoles en línea: https://riptutorial.com/es/spring/topic/3182/creando-y-

usando-frijoles

Capítulo 5: Entendiendo el dispatcher-servlet.xml

Introducción

En Spring Web MVC, la clase DispatcherServlet funciona como el controlador frontal. Es responsable de gestionar el flujo de la aplicación MVC de primavera.

DispatcherServlet también es como servlet normal que debe configurarse en web.xml

Examples

dispatcher-servlet.xml

Este es el archivo de configuración importante donde necesitamos especificar los componentes ViewResolver y View.

El elemento context: component-scan define el paquete base donde DispatcherServlet buscará la clase del controlador.

Aquí, la clase InternalResourceViewResolver se utiliza para ViewResolver.

El prefijo + cadena devuelta por la página controlador + sufijo se invocará para el componente de vista.

Este archivo xml debe estar ubicado dentro del directorio WEB-INF.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:context="http://www.springframework.org/schema/context"
    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
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <context:component-scan base-package="com.srinu.controller.Employee" />

    <bean
        class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix">
            <value>/WEB-INF/views/</value>
        </property>
        <property name="suffix">
            <value>.jsp</value>
        </property>
    </bean>
</beans>
```

Configuración del servlet del despachador en web.xml

En este archivo XML, estamos especificando la clase de servlet DispatcherServlet que actúa como el controlador frontal en Spring Web MVC. Toda la solicitud entrante para el archivo HTML se reenviará al DispatcherServlet.

```
<?xml version="1.0" encoding="UTF-8"?>
<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">
 <servlet>
    <servlet-name>spring</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>spring</servlet-name>
    <url-pattern>*.html</url-pattern>
</servlet-mapping>
</web-app>
```

Lea Entendiendo el dispatcher-servlet.xml en línea:
https://riptutorial.com/es/spring/topic/10092/entendiendo-el-dispatcher-servlet-xml

Capítulo 6: Fuente de la propiedad

Examples

Anotación

Archivo de propiedades de ejemplo: nexus.properties

Ejemplo de contenido del archivo de propiedades:

```
nexus.user=admin
nexus.pass=admin
nexus.rest.uri=http://xxx.xxx.xxx.xxx:xxxx/nexus/service/local/artifact/maven/content
```

Ejemplo de configuración del archivo de contexto xml

```
<context:property-placeholder location="classpath:ReleaseBundle.properties" />
```

Muestra de Property Bean usando anotaciones

```
@Component
@PropertySource(value = { "classpath:nexus.properties" })
public class NexusBean {

    @Value("${" + NexusConstants.NEXUS_USER + "}")
    private String user;

    @Value("${" + NexusConstants.NEXUS_PASS + "}")
    private String pass;

    @Value("${" + NexusConstants.NEXUS_REST_URI + "}")
    private String restUri;
}
```

Muestra de clase constante

```
public class NexusConstants {
    public static final String NexusConstants.NEXUS_USER="";
    public static final String NexusConstants.NEXUS_PASS="";
    public static final String NexusConstants.NEXUS_REST_URI="";
}
```

Ejemplo de configuración xml usando PropertyPlaceholderConfigurer

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations">
     <list>
           <value>classpath:ReleaseBundle.properties</value>
     </list>
</bean>
```

Lea Fuente de la propiedad en línea: https://riptutorial.com/es/spring/topic/6651/fuente-de-la-propiedad

Capítulo 7: Inicialización perezosa de primavera

Examples

Inicialización perezosa en la clase de configuración

```
@Configuration
// @Lazy - For all Beans to load lazily
public class AppConf {

    @Bean
    @Lazy
    public Demo demo() {
        return new Demo();
    }
}
```

Para escaneo de componentes y cableado automático

```
@Component
@Lazy
public class Demo {
    ....
    ....
}

@Component
public class B {

    @Autowired
    @Lazy // If this is not here, Demo will still get eagerly instantiated to satisfy this
request.
    private Demo demo;

    .......
 }
```

Ejemplo de Lazy Init en primavera

`@Lazy` nos permite dar instrucciones al contenedor IOC para retrasar la inicialización de un bean. Por defecto, los beans se crean una instancia tan pronto como se crea el contenedor IOC. `@Lazy` nos permite cambiar este proceso de `@Lazy` instancias.

Lazy-init en primavera es el atributo de la etiqueta bean. Los valores de lazy-init son verdaderos y falsos. Si lazy-init es verdadero, entonces el bean se inicializará cuando se realice una solicitud para el bean. Este bean no se inicializará cuando se inicialice el contenedor de resorte. Si lazy-init es falso, el bean se inicializará con la inicialización del contenedor spring y este es el comportamiento predeterminado.

app-conf.xml

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

<bean id="testA" class="com.concretepage.A"/>
<bean id="testB" class="com.concretepage.B" lazy-init="true"/>
```

A.java

```
package com.concretepage;
public class A {
public A(){
    System.out.println("Bean A is initialized");
   }
}
```

B.java

```
package com.concretepage;
public class B {
public B(){
    System.out.println("Bean B is initialized");
   }
}
```

Prueba de primavera.java

```
package com.concretepage;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class SpringTest {
public static void main(String[] args) {
     ApplicationContext context = new ClassPathXmlApplicationContext("app-conf.xml");
     System.out.println("Feth bean B.");
     context.getBean("testB");
   }
}
```

Salida

```
Bean A is initialized
Feth bean B.
Bean B is initialized
```

Lea Inicialización perezosa de primavera en línea:
https://riptutorial.com/es/spring/topic/6221/inicializacion-perezosa-de-primavera

Capítulo 8: Inyección de dependencia (DI) e Inversión de control (IoC)

Observaciones

El código fuente para aplicaciones de software grandes se organiza típicamente en unidades múltiples. La definición de una unidad normalmente varía según el lenguaje de programación utilizado. Por ejemplo, el código escrito en un lenguaje de programación de procedimientos (como C) está organizado en `functions` o `procedures` . De manera similar, el código en un lenguaje de programación orientado a objetos (como Java, Scala y C #) se organiza en `classes` , `interfaces` , etc. Estas unidades de organización de código se pueden considerar como unidades individuales que constituyen la aplicación de software general.

Cuando las aplicaciones tienen varias unidades, las interdependencias entre esas unidades surgen cuando una unidad tiene que usar otras para completar su funcionalidad. Las unidades dependientes pueden considerarse como `consumers` y las unidades de las que dependen como `providers` de funcionalidad específica.

El enfoque de programación más sencillo es que los consumidores controlen completamente el flujo de una aplicación de software al decidir qué proveedores deben ser instanciados, utilizados y destruidos en qué puntos de la ejecución general de la aplicación. Se dice que los consumidores tienen control total sobre los proveedores durante el flujo de ejecución, que son `dependencies` para los consumidores. En caso de que los proveedores tengan sus propias dependencias, es posible que los consumidores tengan que preocuparse sobre cómo deben inicializarse (y liberarse) los proveedores, lo que hace que el flujo de control sea cada vez más complicado a medida que aumenta la cantidad de unidades en el software. Este enfoque también aumenta el acoplamiento entre unidades, lo que hace cada vez más difícil cambiar las unidades individualmente sin preocuparse por romper otras partes del software.

Inversion of Control (IoC) es un principio de diseño que aboga por la subcontratación de actividades de control de flujo como el descubrimiento, creación de instancias y destrucción de unidades en un marco independiente de los consumidores y proveedores. El principio subyacente detrás de IoC es desacoplar a los consumidores y proveedores, liberando a las unidades de software de tener que preocuparse por descubrir, crear instancias y limpiar sus dependencias y permitir que las unidades se centren en su propia funcionalidad. Este desacoplamiento ayuda a mantener el software extensible y mantenible.

La inyección de dependencia es una de las técnicas para implementar el principio de inversión de control mediante el cual las instancias de dependencias (proveedores) se inyectan en una unidad de software (el consumidor) en lugar de que el consumidor tenga que encontrarlas e instanciarlas.

El framework Spring contiene un módulo de inyección de dependencias en su núcleo que permite que los beans administrados por Spring se inyecten en otros beans administrados por Spring como dependencias.

Examples

Inyectar una dependencia manualmente a través de la configuración XML.

Considera las siguientes clases de Java:

```
class Foo {
  private Bar bar;

  public void foo() {
    bar.baz();
  }
}
```

Como puede verse, la clase `Foo` necesita llamar al método `baz` en una instancia de otra `Bar` clase para que su método `foo` funcione correctamente. Se dice que `Bar` es una dependencia para `Foo` ya que `Foo` no puede funcionar correctamente sin una instancia de `Bar` .

Inyección de constructor

Cuando se utiliza la configuración XML para el framework Spring para definir beans administrados por Spring, un bean de tipo `Foo` se puede configurar de la siguiente manera:

```
<bean class="Foo">
  <constructor-arg>
    <bean class="Bar" />
  </constructor-arg>
</bean>
```

o, alternativamente (más detallado):

```
<bean id="bar" class="bar" />

<bean class="Foo">
  <constructor-arg ref="bar" />
</bean>
```

En ambos casos, Spring Framework primero crea una instancia de `Bar` y la `injects` en una instancia de `Foo` . Este ejemplo asume que la clase `Foo` tiene un constructor que puede tomar una instancia de `Bar` como parámetro, es decir:

```
class Foo {
  private Bar bar;

  public Foo(Bar bar) { this.bar = bar; }
}
```

Este estilo se conoce como **inyección de constructor** porque la dependencia (instancia de `Bar`) se inyecta a través del constructor de clase.

Inyección de propiedad

Otra opción para inyectar la dependencia de `Bar` en `Foo` es:

```
<bean class="Foo">
  <property name="bar">
    <bean class="Bar" />
  </property>
</bean>
```

o, alternativamente (más detallado):

```
<bean id="bar" class="bar" />

<bean class="Foo">
  <property name="bar" ref="bar" />
</bean>
```

Esto requiere que la clase `Foo` tenga un método de establecimiento que acepte una instancia de `Bar` , como:

```
class Foo {
  private Bar bar;

  public void setBar(Bar bar) { this.bar = bar; }
}
```

Inyectando una dependencia manualmente a través de la configuración de Java.

Los mismos ejemplos que se muestran arriba con la configuración XML se pueden volver a escribir con la configuración de Java de la siguiente manera.

Inyección de constructor

```
@Configuration
class AppConfig {
  @Bean
  public Bar bar() { return new Bar(); }

  @Bean
  public Foo foo() { return new Foo(bar()); }
}
```

Inyección de propiedad

```
@Configuration
class AppConfig {
  @Bean
  public Bar bar() { return new Bar(); }

  @Bean
  public Foo foo() {
    Foo foo = new Foo();
    foo.setBar(bar());
```

```
    return foo;
  }
}
```

Autowiring una dependencia a través de la configuración XML

Las dependencias se pueden conectar automáticamente cuando se utiliza la función de escaneo de componentes del framework Spring. Para que el cableado automático funcione, se debe realizar la siguiente configuración XML:

```
<context:annotation-config/>
<context:component-scan base-package="[base package]"/>
```

donde, `base-package` es el paquete Java completamente calificado dentro del cual Spring debe realizar la exploración de componentes.

Inyección de constructor

Las dependencias se pueden inyectar a través del constructor de la clase de la siguiente manera:

```
@Component
class Bar { ... }

@Component
class Foo {
  private Bar bar;

  @Autowired
  public Foo(Bar bar) { this.bar = bar; }
}
```

Aquí, `@Autowired` es una anotación específica de Spring. Spring también admite JSR-299 para habilitar la portabilidad de la aplicación a otros marcos de inyección de dependencias basados en Java. Esto permite que `@Autowired` se reemplace con `@Inject` como:

```
@Component
class Foo {
  private Bar bar;

  @Inject
  public Foo(Bar bar) { this.bar = bar; }
}
```

Inyección de propiedad

Las dependencias también se pueden inyectar utilizando los métodos de establecimiento de la siguiente manera:

```
@Component
class Foo {
  private Bar bar;
```

```
  @Autowired
  public void setBar(Bar bar) { this.bar = bar; }
}
```

Inyección de campo

Autowiring también permite inicializar campos en instancias de clase directamente, de la siguiente manera:

```
@Component
class Foo {
  @Autowired
  private Bar bar;
}
```

Para las versiones 4.1+ de Spring puede usar Opcional para dependencias opcionales.

```
@Component
class Foo {

    @Autowired
    private Optional<Bar> bar;
}
```

El mismo enfoque se puede utilizar para el constructor DI.

```
@Component
class Foo {
    private Optional<Bar> bar;

    @Autowired
    Foo(Optional<Bar> bar) {
        this.bar = bar;
    }
}
```

Autowiring una dependencia a través de la configuración de Java

La inyección del constructor a través de la configuración de Java también puede utilizar el cableado automático, como:

```
@Configuration
class AppConfig {
  @Bean
  public Bar bar() { return new Bar(); }

  @Bean
  public Foo foo(Bar bar) { return new Foo(bar); }
}
```

Lea Inyección de dependencia (DI) e Inversión de control (IoC) en línea:
https://riptutorial.com/es/spring/topic/7295/inyeccion-de-dependencia--di--e-inversion-de-control--

ioc-

Capítulo 9: Lenguaje de Expresión de Primavera (SpEL)

Examples

Referencia de sintaxis

Puede usar `@Value("#{expression}")` para inyectar valor en tiempo de ejecución, en el que la `expression` es una expresión SpEL.

Expresiones literales

Los tipos admitidos incluyen cadenas, fechas, valores numéricos (int, real y hexadecimal), booleano y nulo.

```
"#{'Hello World'}"  //strings
"#{3.1415926}"      //numeric values (double)
"#{true}"           //boolean
"#{null}"           //null
```

Lista en linea

```
"#{1,2,3,4}"              //list of number
"#{{'a','b'},{'x','y'}}"  //list of list
```

Mapas en linea

```
"#{name:'Nikola',dob:'10-July-1856'}"
"#{name:{first:'Nikola',last:'Tesla'},dob:{day:10,month:'July',year:1856}}" //map of maps
```

Métodos de invocación

```
"#{'abc'.length()}"      //evaluates to 3
"#{f('hello')}"   //f is a method in the class to which this expression belongs, it has a
string parameter
```

Lea Lenguaje de Expresión de Primavera (SpEL) en línea:
https://riptutorial.com/es/spring/topic/8109/lenguaje-de-expresion-de-primavera--spel-

Capítulo 10: Núcleo de primavera

Examples

Introducción a Spring Core

Spring es un marco amplio, por lo que Spring Framework se ha dividido en varios módulos que hacen que el resorte sea liviano. Algunos módulos importantes son:

1. Núcleo de primavera
2. Primavera AOP
3. Primavera JDBC
4. Transacción de primavera
5. ORM de primavera
6. Primavera MVC

Todos los módulos de Spring son independientes entre sí, excepto Spring Core. Como Spring Core es el módulo base, en todos los módulos tenemos que usar Spring Core

Núcleo de primavera

Spring Core habla sobre la administración de dependencias. Eso significa que si alguna clase arbitraria proporcionada a Spring entonces Spring puede administrar la dependencia.

¿Qué es una dependencia?
Desde el punto de vista del proyecto, en un proyecto o aplicación hay varias clases con diferentes funcionalidades. y cada clase requiere alguna funcionalidad de otras clases.

Ejemplo:

```
class Engine {

  public void start() {
    System.out.println("Engine started");
  }
}

class Car {

  public void move() {
    // For moving start() method of engine class is required
  }
}
```

Aquí el motor de clase requiere el motor de clase, por lo que podemos decir que el motor de clase depende del automóvil de clase, por lo tanto, en lugar de que nosotros gestionemos esas dependencias por herencia o creando un objeto como punto muerto.

Por herencia:

```
class Engine {

  public void start() {
    System.out.println("Engine started");
  }
}

class Car extends Engine {

  public void move() {
    start(); //Calling super class start method,
  }
}
```

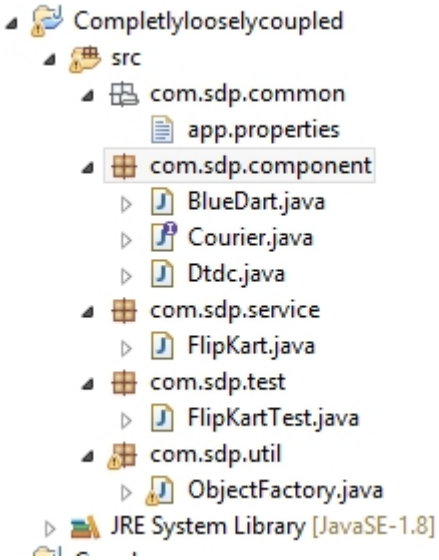
Al crear objeto de clase dependiente:

```
class Engine {

  public void start() {
    System.out.println("Engine started");
  }
}

class Car {

  Engine eng = new Engine();

  public void move() {
   eng.start();
  }
}
```

Entonces, en lugar de administrar la dependencia entre clases, Spring Core toma la responsabilidad de la administración de la dependencia. Pero hay algunas reglas, las clases deben diseñarse con alguna técnica de diseño que sea un patrón de diseño de estrategia.

¿Entendiendo cómo Spring gestiona la dependencia?

Permítanme escribir un fragmento de código que se muestre completamente acoplado, para que pueda comprender fácilmente cómo Spring Core administra la dependencia internamente. Considere un escenario, el negocio en línea de Flipkart está ahí, usa algunas veces el servicio de mensajería DTDC o Blue Dart, así que permítame diseñar una aplicación que muestre una completa sin conexión. El Directorio de Eclipse como barbechos:




```
//Interface
package com.sdp.component;

public interface Courier {
    public String deliver(String iteams,String address);

}
```

// clases de implementación

```
  package com.sdp.component;

public class BlueDart implements Courier {


    public String deliver(String iteams, String address) {

        return iteams+ "Shiped to Address "+address +"Through BlueDart";
    }

}

package com.sdp.component;

public class Dtdc implements Courier {


    public String deliver(String iteams, String address) {
        return iteams+ "Shiped to Address "+address +"Through Dtdc";    }

}
```

// clasificación de componentes

```
package com.sdp.service;

import com.sdp.component.Courier;

public class FlipKart {
    private Courier courier;
```

```
    public void setCourier(Courier courier) {
        this.courier = courier;
    }
    public void shopping(String iteams,String address)
    {
        String status=courier.deliver(iteams, address);
        System.out.println(status);
    }

}
```

// Clases de fabrica para crear y devolver objetos.

```
package com.sdp.util;

import java.io.IOException;
import java.util.Properties;

import com.sdp.component.Courier;

public class ObjectFactory {
private static Properties props;
static{

    props=new Properties();
    try {

props.load(ObjectFactory.class.getClassLoader().getResourceAsStream("com//sdp//common//app.properties")

    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

}
public static Object getInstance(String logicalclassName)
{
    Object obj = null;
    String originalclassName=props.getProperty(logicalclassName);
    try {
         obj=Class.forName(originalclassName).newInstance();
    } catch (InstantiationException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } catch (ClassNotFoundException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
    return obj;
}

    }
```

// archivo de propiedades

```
BlueDart.class=com.sdp.component.BlueDart
Dtdc.class=com.sdp.component.Dtdc
FlipKart.class=com.sdp.service.FlipKart
```

// clase de prueba

```
package com.sdp.test;

import com.sdp.component.Courier;
import com.sdp.service.FlipKart;
import com.sdp.util.ObjectFactory;

public class FlipKartTest {
    public static void main(String[] args) {
        Courier courier=(Courier)ObjectFactory.getInstance("Dtdc.class");
        FlipKart flipkart=(FlipKart)ObjectFactory.getInstance("FlipKart.class");
        flipkart.setCourier(courier);
        flipkart.shopping("Hp Laptop", "SR Nagar,Hyderabad");

    }

}
```

Si escribimos este código, entonces podemos lograr un acoplamiento suelto, esto es aplicable si todas las clases quieren BlueDart o Dtdc, pero si alguna clase quiere BlueDart y alguna otra clase quiere Dtdc, entonces nuevamente estará estrechamente acoplada, así que en lugar de nosotros creando y administrando la inyección de dependencia. Spring Core toma la responsabilidad de crear y administrar los beans. Esperamos. Esto será de utilidad. En el siguiente ejemplo veremos la aplicación! st en Spring Core con datos

Lea Núcleo de primavera en línea: https://riptutorial.com/es/spring/topic/7067/nucleo-de-primavera

Capítulo 11: Obteniendo un SqlRowSet de SimpleJdbcCall

Introducción

Esto describe cómo obtener directamente un **SqlRowSet** utilizando **SimpleJdbcCall** con un procedimiento almacenado en su base de datos que tiene un **parámetro de salida del cursor** ,

Estoy trabajando con una base de datos Oracle. Intenté crear un ejemplo que debería funcionar para otras bases de datos, mi ejemplo de Oracle detalla los problemas con Oracle.

Examples

Creación SimpleJdbcCall

Normalmente, deseará crear su SimpleJdbcCalls en un servicio.

Este ejemplo asume que su procedimiento tiene un único parámetro de salida que es un cursor; tendrá que ajustar sus parámetros de declaración para que coincidan con su procedimiento.

```
@Service
public class MyService() {

@Autowired
    private DataSource dataSource;

    // Autowire your configuration, for example
    @Value("${db.procedure.schema}")
    String schema;

    private SimpleJdbcCall myProcCall;

    // create SimpleJdbcCall after properties are configured
    @PostConstruct
    void initialize() {
        this.myProcCall = new SimpleJdbcCall(dataSource)
                        .withProcedureName("my_procedure_name")
                        .withCatalogName("my_package")
                        .withSchemaName(schema)
                        .declareParameters(new SqlOutParameter(
                            "out_param_name",
                            Types.REF_CURSOR,
                            new SqlRowSetResultSetExtractor()));
    }

    public SqlRowSet myProc() {
        Map<String, Object> out = this.myProcCall.execute();
        return (SqlRowSet) out.get("out_param_name");
    }

}
```

Hay muchas opciones que puedes usar aquí:

- **withoutProcedureColumnMetaDataAccess () es** necesario si ha sobrecargado los nombres de los procedimientos o simplemente no quiere que SimpleJdbcCall se valide en la base de datos.
- **withReturnValue ()** si el procedimiento tiene un valor de retorno. El primer valor dado a declareParameters define el valor de retorno. Además, si su procedimiento es una función, use **withFunctionName** y **executeFunction** al ejecutar.
- **withNamedBinding ()** si desea dar argumentos usando nombres en lugar de una posición.
- **useInParameterNames ()** define el orden de los argumentos. Creo que esto puede ser necesario si pasas tus argumentos como una lista en lugar de un mapa del nombre del argumento al valor. Aunque es posible que solo sea necesario si usas withoutProcedureColumnMetaDataAccess ()

Bases de datos Oracle

Hay una serie de problemas con Oracle. Aquí es cómo resolverlos.

Suponiendo que el parámetro de salida de su procedimiento es el `ref cursor` , obtendrá esta excepción.

> java.sql.SQLException: tipo de columna no válida: 2012

Así que cambie `Types.REF_CURSOR` a `OracleTypes.CURSOR` en **simpleJdbcCall.declareParameters ()**

Apoyo a los tipos de Oracle

Es posible que solo necesite hacer esto si tiene ciertos tipos de columnas en sus datos.

El siguiente problema que encontré fue que los tipos propietarios como `oracle.sql.TIMESTAMPTZ` causaron este error en SqlRowSetResultSetExtractor:

> Tipo de SQL no válido para la columna; la excepción anidada es java.sql.SQLException: tipo de SQL no válido para columna

Por lo tanto, necesitamos crear un **ResultSetExtractor** que admita los tipos de Oracle.
Voy a explicar la razón de la contraseña después de este código.

```
package com.boost.oracle;

import oracle.jdbc.rowset.OracleCachedRowSet;
import org.springframework.dao.DataAccessException;
import org.springframework.jdbc.core.ResultSetExtractor;
import org.springframework.jdbc.support.rowset.ResultSetWrappingSqlRowSet;
import org.springframework.jdbc.support.rowset.SqlRowSet;

import java.sql.ResultSet;
import java.sql.SQLException;

/**
 * OracleTypes can cause {@link org.springframework.jdbc.core.SqlRowSetResultSetExtractor}
```

```
 * to fail due to a Oracle SQL type that is not in the standard {@link java.sql.Types}.
 *
 * Also, types such as {@link oracle.sql.TIMESTAMPTZ} require a Connection when processing
 * the ResultSet; {@link OracleCachedRowSet#getConnectionInternal()} requires a JNDI
 * DataSource name or the username and password to be set.
 *
 * For now I decided to just set the password since changing SpringBoot to a JNDI DataSource
 * configuration is a bit complicated.
 *
 * Created by Arlo White on 2/23/17.
 */
public class OracleSqlRowSetResultSetExtractor implements ResultSetExtractor<SqlRowSet> {

    private String oraclePassword;

    public OracleSqlRowSetResultSetExtractor(String oraclePassword) {
        this.oraclePassword = oraclePassword;
    }

    @Override
    public SqlRowSet extractData(ResultSet rs) throws SQLException, DataAccessException {
        OracleCachedRowSet cachedRowSet = new OracleCachedRowSet();
        // allows getConnectionInternal to get a Connection for TIMESTAMPTZ
        cachedRowSet.setPassword(oraclePassword);
        cachedRowSet.populate(rs);
        return new ResultSetWrappingSqlRowSet(cachedRowSet);
    }

}
```

Ciertos tipos de Oracle requieren una conexión para obtener el valor de columna de un ResultSet. TIMESTAMPTZ es uno de estos tipos. Entonces, cuando se `rowSet.getTimestamp(colIndex)` , obtendrás esta excepción:

> Causado por: java.sql.SQLException: Una o más de las propiedades de autentificación de RowSet no establecidas en oracle.jdbc.rowset.OracleCachedRowSet.getConnectionInternal (OracleCachedRowSp.P.P.P.). : 3717) en org.springframework.jdbc.support.rowset.ResultSetWrappingSqlRowSet.getTimestamp

Si profundiza en este código, verá que OracleCachedRowSet necesita la contraseña o un nombre de JNDI DataSource para obtener una conexión. Si prefiere la búsqueda JNDI, simplemente verifique que OracleCachedRowSet tenga configurado DataSourceName.

Así que en mi Servicio, yo Autowire en la contraseña y declaro el parámetro de salida así:

```
new SqlOutParameter("cursor_param_name", OracleTypes.CURSOR, new
OracleSqlRowSetResultSetExtractor(oraclePassword))
```

Lea Obteniendo un SqlRowSet de SimpleJdbcCall en línea:
https://riptutorial.com/es/spring/topic/9235/obteniendo-un-sqlrowset-de-simplejdbccall

Capítulo 12: Perfil de primavera

Examples

Spring Profiles permite configurar partes disponibles para ciertos entornos.

Cualquier `@Component` o `@Configuration` podría estar marcado con la anotación de `@Profile`

```
@Configuration
@Profile("production")
public class ProductionConfiguration {

    // ...
}
```

Lo mismo en la configuración XML.

```
<beans profile="dev">
    <bean id="dataSource" class="<some data source class>" />
</beans>
```

Los perfiles activos se podrían configurar en el archivo `application.properties`

```
spring.profiles.active=dev,production
```

o especificado desde la línea de comando

```
--spring.profiles.active=dev,hsqldb
```

o en SpringBoot

```
SpringApplication.setAdditionalProfiles("dev");
```

Es posible habilitar perfiles en Pruebas usando la anotación `@ActiveProfiles("dev")`

Lea Perfil de primavera en línea: https://riptutorial.com/es/spring/topic/9981/perfil-de-primavera

Capítulo 13: Plantilla de descanso

Examples

Descarga de un archivo grande

Los métodos `getForObject` y `getForEntity` de `RestTemplate` cargan la respuesta completa en la memoria. Esto no es adecuado para descargar archivos grandes, ya que puede causar excepciones de memoria. Este ejemplo muestra cómo transmitir la respuesta de una solicitud GET.

```
RestTemplate restTemplate // = ...;

// Optional Accept header
RequestCallback requestCallback = request -> request.getHeaders()
        .setAccept(Arrays.asList(MediaType.APPLICATION_OCTET_STREAM, MediaType.ALL));

// Streams the response instead of loading it all in memory
ResponseExtractor<Void> responseExtractor = response -> {
    // Here I write the response to a file but do what you like
    Path path = Paths.get("some/path");
    Files.copy(response.getBody(), path);
    return null;
};
restTemplate.execute(URI.create("www.something.com"), HttpMethod.GET, requestCallback,
responseExtractor);
```

Tenga en cuenta que no puede simplemente devolver el `InputStream` desde el extractor, porque en el momento en que el método de ejecución devuelve, la conexión subyacente y la secuencia ya están cerradas.

Uso de la autenticación básica preventiva con RestTemplate y HttpClient

La autenticación básica preventiva es la práctica de enviar las credenciales de autenticación básica de HTTP (nombre de usuario y contraseña) *antes de que* un servidor responda con una respuesta `401` solicitándolas. Esto puede ahorrar una solicitud de ida y vuelta cuando se consumen apis REST que se sabe que requieren una autenticación básica.

Como se describe en la documentación de Spring , Apache HttpClient puede usarse como la implementación subyacente para crear solicitudes HTTP mediante el uso de `HttpComponentsClientHttpRequestFactory` . HttpClient se puede configurar para realizar una autenticación básica preventiva .

La siguiente clase extiende `HttpComponentsClientHttpRequestFactory` para proporcionar autenticación básica preferente.

```
/**
 * {@link HttpComponentsClientHttpRequestFactory} with preemptive basic
 * authentication to avoid the unnecessary first 401 response asking for
```

```
 * credentials.
 * <p>
 * Only preemptively sends the given credentials to the given host and
 * optionally to its subdomains. Matching subdomains can be useful for APIs
 * using multiple subdomains which are not always known in advance.
 * <p>
 * Other configurations of the {@link HttpClient} are not modified (e.g. the
 * default credentials provider).
 */
public class PreAuthHttpComponentsClientHttpRequestFactory extends
HttpComponentsClientHttpRequestFactory {

    private String hostName;
    private boolean matchSubDomains;
    private Credentials credentials;

    /**
     * @param httpClient
     *            client
     * @param hostName
     *            host name
     * @param matchSubDomains
     *            whether to match the host's subdomains
     * @param userName
     *            basic authentication user name
     * @param password
     *            basic authentication password
     */
    public PreAuthHttpComponentsClientHttpRequestFactory(HttpClient httpClient, String
hostName,
            boolean matchSubDomains, String userName, String password) {
        super(httpClient);
        this.hostName = hostName;
        this.matchSubDomains = matchSubDomains;
        credentials = new UsernamePasswordCredentials(userName, password);
    }

    @Override
    protected HttpContext createHttpContext(HttpMethod httpMethod, URI uri) {
        // Add AuthCache to the execution context
        HttpClientContext context = HttpClientContext.create();
        context.setCredentialsProvider(new PreAuthCredentialsProvider());
        context.setAuthCache(new PreAuthAuthCache());
        return context;
    }

    /**
     * @param host
     *            host name
     * @return whether the configured credentials should be used for the given
     *         host
     */
    protected boolean hostNameMatches(String host) {
        return host.equals(hostName) || (matchSubDomains && host.endsWith("." + hostName));
    }

    private class PreAuthCredentialsProvider extends BasicCredentialsProvider {
        @Override
        public Credentials getCredentials(AuthScope authscope) {
            if (hostNameMatches(authscope.getHost())) {
                // Simulate a basic authenticationcredentials entry in the
```

```
                // credentials provider.
                return credentials;
            }
            return super.getCredentials(authscope);
        }
    }

    private class PreAuthAuthCache extends BasicAuthCache {
        @Override
        public AuthScheme get(HttpHost host) {
            if (hostNameMatches(host.getHostName())) {
                // Simulate a cache entry for this host. This instructs
                // HttpClient to use basic authentication for this host.
                return new BasicScheme();
            }
            return super.get(host);
        }
    }
}
```

Esto se puede utilizar de la siguiente manera:

```
HttpClientBuilder builder = HttpClientBuilder.create();
ClientHttpRequestFactory requestFactory =
    new PreAuthHttpComponentsClientHttpRequestFactory(builder.build(),
        "api.some-host.com", true, "api", "my-key");
RestTemplate restTemplate = new RestTemplate(requestFactory);
```

Usando la autenticación básica con HttpComponent's HttpClient

El uso de `HttpClient` como la implementación subyacente de `RestTemplate` para crear solicitudes HTTP permite el manejo automático de las solicitudes de autenticación básicas (una respuesta http 401) cuando se interactúa con las API. Este ejemplo muestra cómo configurar un `RestTemplate` para lograr esto.

```
// The credentials are stored here
CredentialsProvider credsProvider = new BasicCredentialsProvider();
credsProvider.setCredentials(
        // AuthScope can be configured more extensively to restrict
        // for which host/port/scheme/etc the credentials will be used.
        new AuthScope("somehost", AuthScope.ANY_PORT),
        new UsernamePasswordCredentials("username", "password"));

// Use the credentials provider
HttpClientBuilder builder = HttpClientBuilder.create();
builder.setDefaultCredentialsProvider(credsProvider);

// Configure the RestTemplate to use HttpComponent's HttpClient
ClientHttpRequestFactory requestFactory =
        new HttpComponentsClientHttpRequestFactory(builder.build());
RestTemplate restTemplate = new RestTemplate(requestFactory);
```

Configuración de encabezados en Spring RestTemplate request

Los métodos de `exchange` de `RestTemplate` permiten especificar una `HttpEntity` que se escribirá en

la solicitud cuando se ejecute el método. Puede agregar encabezados (como agente de usuario, referente ...) a esta entidad:

```
public void testHeader(final RestTemplate restTemplate){
        //Set the headers you need send
        final HttpHeaders headers = new HttpHeaders();
        headers.set("User-Agent", "eltabo");

        //Create a new HttpEntity
        final HttpEntity<String> entity = new HttpEntity<String>(headers);

        //Execute the method writing your HttpEntity to the request
        ResponseEntity<Map> response = restTemplate.exchange("https://httpbin.org/user-agent",
HttpMethod.GET, entity, Map.class);
        System.out.println(response.getBody());
    }
```

También puede agregar un interceptor a su `RestTemplate` si necesita agregar los mismos encabezados a múltiples solicitudes:

```
public void testHeader2(final RestTemplate restTemplate){
    //Add a ClientHttpRequestInterceptor to the RestTemplate
    restTemplate.getInterceptors().add(new ClientHttpRequestInterceptor(){
        @Override
        public ClientHttpResponse intercept(HttpRequest request, byte[] body,
ClientHttpRequestExecution execution) throws IOException {
            request.getHeaders().set("User-Agent", "eltabo");//Set the header for each request
            return execution.execute(request, body);
        }
    });

    ResponseEntity<Map> response = restTemplate.getForEntity("https://httpbin.org/user-agent",
Map.class);
    System.out.println(response.getBody());

    ResponseEntity<Map> response2 = restTemplate.getForEntity("https://httpbin.org/headers",
Map.class);
    System.out.println(response2.getBody());
}
```

Resultados genéricos de Spring RestTemplate

Para que `RestTemplate` comprenda el contenido genérico del contenido devuelto, debemos definir la referencia de tipo de resultado.

`org.springframework.core.ParameterizedTypeReference` se ha introducido desde 3.2

```
Wrapper<Model> response = restClient.exchange(url,
                          HttpMethod.GET,
                          null,
                          new ParameterizedTypeReference<Wrapper<Model>>() {}).getBody();
```

Podría ser útil para obtener, por ejemplo, la `List<User>` de un controlador.

Lea Plantilla de descanso en línea: https://riptutorial.com/es/spring/topic/5896/plantilla-de-

descanso

Capítulo 14: Plantilla Jdbc

Introducción

La clase JdbcTemplate ejecuta consultas SQL, sentencias de actualización y llamadas a procedimientos almacenados, realiza iteraciones sobre ResultSets y extracción de valores de parámetros devueltos. También captura las excepciones de JDBC y las traduce a la jerarquía de excepciones genérica, más informativa, definida en el paquete org.springframework.dao.

Las instancias de la clase JdbcTemplate son seguras para subprocesos una vez configuradas para que pueda inyectar de forma segura esta referencia compartida en varios DAO.

Examples

Métodos básicos de consulta

Algunos de los métodos queryFor * disponibles en JdbcTemplate son útiles para sentencias de SQL simples que realizan operaciones CRUD.

Pidiendo fecha

```
String sql = "SELECT create_date FROM customer WHERE customer_id = ?";
int storeId = jdbcTemplate.queryForObject(sql, java.util.Date.class, customerId);
```

Consultar por entero

```
String sql = "SELECT store_id FROM customer WHERE customer_id = ?";
int storeId = jdbcTemplate.queryForObject(sql, Integer.class, customerId);
```

O

```
String sql = "SELECT store_id FROM customer WHERE customer_id = ?";
int storeId = jdbcTemplate.queryForInt(sql, customerId);                //Deprecated in
spring-jdbc 4
```

Consultar por Cadena

```
String sql = "SELECT first_Name FROM customer WHERE customer_id = ?";
String firstName = jdbcTemplate.queryForObject(sql, String.class, customerId);
```

Consultar por lista

```
String sql = "SELECT first_Name FROM customer WHERE store_id =  ?";
List<String> firstNameList = jdbcTemplate.queryForList(sql, String.class, storeId);
```

Consulta de lista de mapas

```
int storeId = 1;
DataSource dataSource = ... //
JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
String sql = "SELECT * FROM customer WHERE store_id = ?";
List<Map<String, Object>> mapList = jdbcTemplate.queryForList(sql, storeId);

for(Map<String, Object> entryMap : mapList)
{
  for(Entry<String, Object> entry : entryMap.entrySet())
  {
       System.out.println(entry.getKey() + " / " + entry.getValue());
  }
  System.out.println("---");
}
```

SQLRowSet

```
DataSource dataSource = ... //
JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
String sql = "SELECT * FROM customer";
SqlRowSet rowSet = jdbcTemplate.queryForRowSet(sql);

while(rowSet.next())
{
  String firstName = rowSet.getString("first_name");
  String lastName = rowSet.getString("last_name");
  System.out.println("Vorname: " + firstName);
  System.out.println("Nachname: " + lastName);
  System.out.println("---”);
}
```

O

```
String sql = "SELECT * FROM customer";
List<Customer> customerList = jdbcTemplate.query(sql, new RowMapper<Customer>()    {

  @Override
  public Customer mapRow(ResultSet rs, int rowNum) throws SQLException
  {
    Customer customer = new Customer();
    customer.setFirstName(rs.getString("first_Name"));
    customer.setLastName(rs.getString("first_Name"));
    customer.setEmail(rs.getString("email"));

    return customer;
  }

});
```

Operaciones por lotes

JdbcTemplate también proporciona métodos convenientes para ejecutar operaciones por lotes.

Insertar por lotes

```
final ArrayList<Student> list = // Get list of students to insert..
String sql = "insert into student (id, f_name, l_name, age, address) VALUES (?, ?, ?, ?, ?)"
jdbcTemplate.batchUpdate(sql, new BatchPreparedStatementSetter(){
    @Override
    public void setValues(PreparedStatement ps, int i) throws SQLException {
        Student s = l.get(i);
        ps.setString(1, s.getId());
        ps.setString(2, s.getF_name());
        ps.setString(3, s.getL_name());
        ps.setInt(4, s.getAge());
        ps.setString(5, s.getAddress());
    }

    @Override
    public int getBatchSize() {
        return l.size();
    }
});
```

Actualización por lotes

```
final ArrayList<Student> list = // Get list of students to update..
String sql = "update student set f_name = ?, l_name = ?, age = ?, address = ? where id = ?"
jdbcTemplate.batchUpdate(sql, new BatchPreparedStatementSetter(){
    @Override
    public void setValues(PreparedStatement ps, int i) throws SQLException {
        Student s = l.get(i);
        ps.setString(1, s.getF_name());
        ps.setString(2, s.getL_name());
        ps.setInt(3, s.getAge());
        ps.setString(4, s.getAddress());
        ps.setString(5, s.getId());
    }

    @Override
    public int getBatchSize() {
        return l.size();
    }
});
```

Hay otros métodos batchUpdate que aceptan Lista de matriz de objetos como parámetros de entrada. Estos métodos utilizan internamente BatchPreparedStatementSetter para establecer los valores de la lista de matrices en la declaración de SQL.

NamedParameterJdbcTemplate extensión de JdbcTemplate

> La clase `NamedParameterJdbcTemplate` agrega soporte para la programación de sentencias JDBC utilizando parámetros nombrados, a diferencia de la programación de sentencias JDBC utilizando solo argumentos clásicos de marcador de posición ('?'). La clase `NamedParameterJdbcTemplate` envuelve un `JdbcTemplate` , y delega a la envuelta `JdbcTemplate` para hacer gran parte de su trabajo.

```
DataSource dataSource = ... //
NamedParameterJdbcTemplate jdbcTemplate = new NamedParameterJdbcTemplate(dataSource);
```

```
String sql = "SELECT count(*) FROM customer WHERE city_name=:cityName";
Map<String, String> params = Collections.singletonMap("cityName", cityName);
int count = jdbcTemplate.queryForObject(sql, params, Integer.class);
```

Lea Plantilla Jdbc en línea: https://riptutorial.com/es/spring/topic/7742/plantilla-jdbc

Capítulo 15: Registro condicional de frijol en primavera.

Observaciones

Punto importante a tener en cuenta al usar la condición

- La clase de condición se denomina clase directa (no frijol de primavera), por lo **que no puede** usar la inyección de la propiedad `@Value` , es decir, no se pueden inyectar otros frijoles de primavera dentro de ella.
- Desde documentos java: las *condiciones deben seguir las mismas restricciones que* `BeanFactoryPostProcessor` *y tener cuidado de nunca interactuar con las instancias de bean* . Las restricciones a las que se hace referencia aquí son *A* `BeanFactoryPostProcessor` *puede interactuar con y modificar definiciones de bean, pero nunca instancias de bean. Hacerlo puede provocar una instanciación prematura del frijol, violar el contenedor y causar efectos secundarios no deseados.*

Examples

Registrar beans solo cuando se especifica una propiedad o valor

Un grano de primavera se puede configurar de tal manera que se registrará *sólo* si tiene un valor particular *o* un determinado inmueble se cumple. Para hacerlo, implemente `Condition.matches` para verificar la propiedad / valor:

```
public class PropertyCondition implements Condition {
    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
        return context.getEnvironment().getProperty("propertyName") != null;
        // optionally check the property value
    }
}
```

En la configuración de Java, use la implementación anterior como condición para registrar el bean. Tenga en cuenta el uso de la anotación @Conditional .

```
@Configuration
public class MyAppConfig {

    @Bean
    @Conditional(PropertyCondition.class)
    public MyBean myBean() {
      return new MyBean();
    }
}
```

En `PropertyCondition` , se puede evaluar cualquier número de condiciones. Sin embargo, se

recomienda separar la implementación de cada condición para mantenerlas acopladas de manera flexible. Por ejemplo:

```
@Configuration
public class MyAppConfig {

    @Bean
    @Conditional({PropertyCondition.class, SomeOtherCondition.class})
    public MyBean myBean() {
      return new MyBean();
    }
}
```

Anotaciones de condición

Excepto en la anotación principal de `@conditional` hay un conjunto de anotaciones similares para usar en diferentes casos.

Condiciones de clase

Las anotaciones `@ConditionalOnClass` y `@ConditionalOnMissingClass` permiten que la configuración se incluya en función de la presencia o ausencia de clases específicas.

Por ejemplo, cuando `OObjectDatabaseTx.class` se agrega a las dependencias y no hay ningún bean `OrientWebConfigurer` , creamos el configurador.

```
@Bean
@ConditionalOnWebApplication
@ConditionalOnClass(OObjectDatabaseTx.class)
@ConditionalOnMissingBean(OrientWebConfigurer.class)
public OrientWebConfigurer orientWebConfigurer() {
    return new OrientWebConfigurer();
}
```

Condiciones de frijol

Las anotaciones `@ConditionalOnBean` y `@ConditionalOnMissingBean` permiten `@ConditionalOnMissingBean` un bean en función de la presencia o ausencia de beans específicos. Puede usar el atributo de valor para especificar beans por tipo, o nombre para especificar beans por nombre. El atributo de búsqueda le permite limitar la jerarquía de `ApplicationContext` que debe considerarse al buscar beans.

Vea el ejemplo anterior cuando verificamos si no hay un bean definido.

Condiciones de propiedad

La anotación `@ConditionalOnProperty` permite que la configuración se incluya en base a una

propiedad de Spring Environment. Use los atributos de prefijo y nombre para especificar la propiedad que debe verificarse. Por defecto, cualquier propiedad que exista y que no sea igual a `false` coincidirá. También puede crear verificaciones más avanzadas utilizando los atributos `havingValue` y `matchIfMissing` .

```
@ConditionalOnProperty(value='somebean.enabled', matchIfMissing = true, havingValue="yes")
@Bean
public SomeBean someBean(){
}
```

Condiciones de recursos

La anotación `@ConditionalOnResource` permite que la configuración se incluya solo cuando un recurso específico está presente.

```
@ConditionalOnResource(resources = "classpath:init-db.sql")
```

Condiciones de la aplicación web

Las anotaciones `@ConditionalOnWebApplication` y `@ConditionalOnNotWebApplication` permiten que se incluya la configuración dependiendo de si la aplicación es una 'aplicación web'.

```
@Configuration
@ConditionalOnWebApplication
public class MyWebMvcAutoConfiguration {...}
```

Condiciones de expresión SpEL

La anotación `@ConditionalOnExpression` permite incluir la configuración en función del resultado de una expresión SpEL.

```
@ConditionalOnExpression("${rest.security.enabled}==false")
```

Lea Registro condicional de frijol en primavera. en línea: https://riptutorial.com/es/spring/topic/4732/registro-condicional-de-frijol-en-primavera-

Capítulo 16: SOAP WS DE CONSUMO

Examples

Consumiendo un SOAP WS con autenticación básica

Crea tu propio WSMessageSender:

```
import java.io.IOException;
import java.net.HttpURLConnection;

import org.springframework.ws.transport.http.HttpUrlConnectionMessageSender;

import sun.misc.BASE64Encoder;

public class CustomWSMessageSender extends HttpUrlConnectionMessageSender{

    @Override
    protected void prepareConnection(HttpURLConnection connection)
            throws IOException {

        BASE64Encoder enc = new sun.misc.BASE64Encoder();
        String userpassword = "yourUser:yourPassword";
        String encodedAuthorization = enc.encode( userpassword.getBytes() );
        connection.setRequestProperty("Authorization", "Basic " + encodedAuthorization);

        super.prepareConnection(connection);
    }
}
```

En su clase de configuración WS, configure el MessageSender que acaba de crear:

```
myWSClient.setMessageSender(new CustomWSMessageSender());
```

Lea SOAP WS DE CONSUMO en línea: https://riptutorial.com/es/spring/topic/9451/soap-ws-de-consumo

Capítulo 17: Tarea de Ejecución y Programación

Examples

Habilitar la programación

Spring proporciona un soporte útil para la programación de tareas. Para activarlo, simplemente anotaciones en cualquiera de sus `@Configuration` clases con `@EnableScheduling` :

```
@Configuration
@EnableScheduling
public class MyConfig {

    // Here it goes your configuration
}
```

Retraso fijo

Si queremos que algún código se ejecute periódicamente después de que finalice la ejecución anterior, deberíamos usar un retraso fijo (medido en milisegundos):

```
@Component
public class MyScheduler{

    @Scheduled(fixedDelay=5000)
    public void doSomething() {
        // this will execute periodically, after the one before finishes
    }
}
```

Tipo de interés fijo

Si queremos que se ejecute algo periódicamente, este código se activará una vez por el valor en milisegundos que especificamos:

```
@Component
public class MyScheduler{

    @Scheduled(fixedRate=5000)
    public void doSomething() {
        // this will execute periodically
    }
}
```

Cron expresión

Una expresión cron consta de seis campos secuenciales:

```
second, minute, hour, day of month, month, day(s) of week
```

y se declara como sigue

```
@Scheduled(cron = "* * * * * *")
```

También podemos establecer la zona horaria como -

```
@Scheduled(cron="* * * * * *", zone="Europe/Istanbul")
```

Notas: -

```
syntax            means               example               explanation
-------------------------------------------------------------------------------
*                 match any           "* * * * * *"         do always
*/x               every x             "*/5 * * * * *"       do every five seconds
?                 no specification    "0 0 0 25 12 ?"       do every Christmas Day
```

Ejemplo: -

```
syntax                       means
-------------------------------------------------------------------------------
"0 0 * * * *"                the top of every hour of every day.
"*/10 * * * * *"             every ten seconds.
"0 0 8-10 * * *"             8, 9 and 10 o'clock of every day.
"0 0/30 8-10 * * *"          8:00, 8:30, 9:00, 9:30 and 10 o'clock every day.
"0 0 9-17 * * MON-FRI"       on the hour nine-to-five weekdays
"0 0 0 25 12 ?"              every Christmas Day at midnight
```

Un método declarado con `@Scheduled()` se llama explícitamente para cada caso coincidente.

Si queremos que se ejecute algún código cuando se cumpla una expresión cron, entonces tenemos que especificarlo en la anotación:

```
@Component
public class MyScheduler{

    @Scheduled(cron="*/5 * * * * MON-FRI")
    public void doSomething() {
        // this will execute on weekdays
    }
}
```

Si queremos imprimir la hora actual en nuestra consola para cada 5 segundos,

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;

import java.text.SimpleDateFormat;
import java.util.Date;
```

```
@Component
public class Scheduler {

    private static final Logger log = LoggerFactory.getLogger(Scheduler.class);
    private static final SimpleDateFormat dateFormat = new SimpleDateFormat("HH:mm:ss");

    @Scheduled(cron = "*/5 * * * * *")
    public void currentTime() {
        log.info("Current Time      = {}", dateFormat.format(new Date()));
    }

}
```

Ejemplo utilizando la configuración XML:

Ejemplo de clase:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;

import java.text.SimpleDateFormat;
import java.util.Date;


@Component("schedulerBean")
public class Scheduler {

    private static final Logger log = LoggerFactory.getLogger(Scheduler.class);
    private static final SimpleDateFormat dateFormat = new SimpleDateFormat("HH:mm:ss");

    public void currentTime() {
        log.info("Current Time      = {}", dateFormat.format(new Date()));
    }

}
```

Ejemplo XML (task-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"
    xmlns:task="http://www.springframework.org/schema/task"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-4.1.xsd
        http://www.springframework.org/schema/task
        http://www.springframework.org/schema/task/spring-task-4.1.xsd">


    <task:scheduled-tasks scheduler="scheduledTasks">
        <task:scheduled ref="schedulerBean" method="currentTime" cron="*/5 * * * * MON-FRI" />
    </task:scheduled-tasks>

    <task:scheduler id="scheduledTasks" />

</beans>
```

Lea Tarea de Ejecución y Programación en línea:
https://riptutorial.com/es/spring/topic/6080/tarea-de-ejecucion-y-programacion

Capítulo 18: Validación de frijol Spring JSR 303

Introducción

Spring tiene soporte de validación de bean JSR303. Podemos usar esto para hacer validación de bean de entrada. Separe la lógica de validación de la lógica de negocios utilizando JSR303.

Examples

JSR303 Validación basada en anotaciones en ejemplos de resortes

Agregue cualquier implementación JSR 303 a su classpath. El popular utilizado es el validador Hibernate de Hibernate.

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>4.2.0.Final</version>
</dependency>
```

Digamos que hay un api de descanso para crear usuarios en el sistema

```
@RequestMapping(value="/registeruser", method=RequestMethod.POST)
public String registerUser(User user);
```

La muestra json de entrada se vería como se muestra a continuación

```
{"username" : "abc@abc.com", "password" : "password1", "password2":"password1"}
```

Usuario.java

```
public class User {

    private String username;
    private String password;
    private String password2;

    getXXX and setXXX

}
```

Podemos definir las validaciones JSR 303 en la clase de usuario como se muestra a continuación.

```
public class User {
```

```
    @NotEmpty
    @Size(min=5)
    @Email
    private String username;

    @NotEmpty
    private String password;

    @NotEmpty
    private String password2;

}
```

También es posible que debamos tener un validador de negocios, como la contraseña y la contraseña2 (confirmar contraseña) son las mismas, para esto podemos agregar un validador personalizado como se muestra a continuación. Escriba una anotación personalizada para anotar el campo de datos.

```
@Target({ ElementType.FIELD })
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = PasswordValidator.class)
public @interface GoodPassword {
    String message() default "Passwords wont match.";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

Escribe una clase de Validador para aplicar la lógica de Validación.

```
public class PastValidator implements ConstraintValidator<GoodPassword, User> {
    @Override
    public void initialize(GoodPassword annotation) {}

    @Override
    public boolean isValid(User user, ConstraintValidatorContext context) {
        return user.getPassword().equals(user.getPassword2());
    }
}
```

Agregando esta validación a la clase de usuario

```
@GoodPassword
public class User {

    @NotEmpty
    @Size(min=5)
    @Email
    private String username;

    @NotEmpty
    private String password;

    @NotEmpty
    private String password2;
}
```

@Valid activa la validación en Spring. BindingResult es un objeto inyectado por spring que tiene una lista de errores después de la validación.

```
public String registerUser(@Valid User user, BindingResult result);
```

La anotación JSR 303 tiene atributos de mensaje que se pueden usar para proporcionar mensajes personalizados.

```
@GoodPassword
public class User {

    @NotEmpty(message="Username Cant be empty")
    @Size(min=5, message="Username cant be les than 5 chars")
    @Email(message="Should be in email format")
    private String username;

    @NotEmpty(message="Password cant be empty")
    private String password;

    @NotEmpty(message="Password2 cant be empty")
    private String password2;

}
```

Spring JSR 303 Validation - Personaliza los mensajes de error

Supongamos que tenemos una clase simple con anotaciones de validación

```
public class UserDTO {
    @NotEmpty
    private String name;

    @Min(18)
    private int age;

//getters/setters
}
```

Un controlador para comprobar la validez de UserDTO.

```
@RestController
public class ValidationController {

    @RequestMapping(value = "/validate", method = RequestMethod.POST)
    public ResponseEntity<String> check(@Valid @RequestBody UserDTO userDTO,
           BindingResult bindingResult) {
        return new ResponseEntity<>("ok" , HttpStatus.OK);
    }
}
```

Y una prueba.

```
@Test
public void testValid() throws Exception {
```

```
    TestRestTemplate template = new TestRestTemplate();
    String url = base + contextPath + "/validate";
    Map<String, Object> params = new HashMap<>();
    params.put("name", "");
    params.put("age", "10");

    MultiValueMap<String, String> headers = new LinkedMultiValueMap<>();
    headers.add("Content-Type", "application/json");

    HttpEntity<Map<String, Object>> request = new HttpEntity<>(params, headers);
    String res = template.postForObject(url, request, String.class);

    assertThat(res, equalTo("ok"));
}
```

Tanto el nombre como la edad no son válidos, por lo que en BindingResult tenemos dos errores de validación. Cada uno tiene una matriz de códigos.

Códigos para el control mínimo

```
0 = "Min.userDTO.age"
1 = "Min.age"
2 = "Min.int"
3 = "Min"
```

Y para el cheque NotEmpty

```
0 = "NotEmpty.userDTO.name"
1 = "NotEmpty.name"
2 = "NotEmpty.java.lang.String"
3 = "NotEmpty"
```

Agreguemos un archivo custom.properties para sustituir los mensajes predeterminados.

```
@SpringBootApplication
@Configuration
public class DemoApplication {

    @Bean(name = "messageSource")
    public MessageSource messageSource() {
        ReloadableResourceBundleMessageSource bean = new
ReloadableResourceBundleMessageSource();
        bean.setBasename("classpath:custom");
        bean.setDefaultEncoding("UTF-8");
        return bean;
    }

    @Bean(name = "validator")
    public LocalValidatorFactoryBean validator() {
        LocalValidatorFactoryBean bean = new LocalValidatorFactoryBean();
        bean.setValidationMessageSource(messageSource());
        return bean;
    }

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

```
}
```

Si agregamos al archivo custom.properties la línea

```
NotEmpty=The field must not be empty!
```

El nuevo valor se muestra para el error. Para resolver el validador de mensajes, revise los códigos que comienzan desde el principio para encontrar los mensajes adecuados.

Por lo tanto, cuando definimos la clave `NotEmpty` en el archivo .properties para todos los casos donde se `@NotEmpty` anotación `@NotEmpty` se aplica nuestro mensaje.

Si definimos un mensaje.

```
Min.int=Some custom message here.
```

Todas las anotaciones donde aplicamos min check a valores enteros utilizan el mensaje recién definido.

La misma lógica podría aplicarse si necesitamos localizar los mensajes de error de validación.

Uso de @Valid para validar POJOs anidados

Supongamos que tenemos un usuario de clase POJO que necesitamos validar.

```
public class User {

    @NotEmpty
    @Size(min=5)
    @Email
    private String email;
}
```

y un método de controlador para validar la instancia de usuario

```
public String registerUser(@Valid User user, BindingResult result);
```

Extendamos al usuario con una dirección POJO anidada que también necesitamos validar.

```
public class Address {

    @NotEmpty
    @Size(min=2, max=3)
    private String countryCode;
}
```

Solo agregue la anotación `@Valid` en el campo de dirección para ejecutar la validación de los POJOs anidados.

```
public class User {
```

```
    @NotEmpty
    @Size(min=5)
    @Email
    private String email;

    @Valid
    private Address address;
}
```

Lea Validación de frijol Spring JSR 303 en línea:
https://riptutorial.com/es/spring/topic/9882/validacion-de-frijol-spring-jsr-303

Creditos

S. No	Capítulos	Contributors
1	Empezando con la primavera	Andrei Maieras, Community, dimitrisli, guille11, Hitesh Kumar, ipsi, Panther, Rajanikanta Pradhan
2	Alcances de frijol	Constantine, Panther, Taylor, Tim Tong, xpadro
3	Configuración de ApplicationContext	nicholas.hauschild
4	Creando y usando frijoles	CollinD, Constantine, Harshal Patil, JamesENL, Maciej Walkowiak, mszymborski, nicholas.hauschild, Panther, StanislavL, Stefan Isele - prefabware.com, Tim Tong
5	Entendiendo el dispatcher-servlet.xml	Srinivas Gadilli
6	Fuente de la propiedad	Gautam Jose, Panther
7	Inicialización perezosa de primavera	AdamIJK, DavidR, Moshe Arad
8	Inyección de dependencia (DI) e Inversión de control (IoC)	manish, Sergii Bishyr, walsh
9	Lenguaje de Expresión de Primavera (SpEL)	Stephen Leppik, walsh
10	Núcleo de primavera	dimitrisli, JDC, parlad neupane, Rajanikanta Pradhan
11	Obteniendo un SqlRowSet de SimpleJdbcCall	Arlo
12	Perfil de primavera	StanislavL
13	Plantilla de descanso	bernie, eltabo, StanislavL
14	Plantilla Jdbc	Setu, smichel, StanislavL

15	Registro condicional de frijol en primavera.	4444, Bond - Java Bond, StanislavL
16	SOAP WS DE CONSUMO	guille11
17	Tarea de Ejecución y Programación	guille11, Johir, Xtreme Biker
18	Validación de frijol Spring JSR 303	Praneeth Ramesh, StanislavL