# APPRENEZ
# spring

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#spring

# Table des matières

À propos ... 1

Chapitre 1: Commencer avec le printemps ... 2

Remarques ... 2

Versions ... 2

Examples ... 2

Configuration (configuration XML) ... 2

Présentation des fonctionnalités de base du ressort par exemple ... 3

La description ... 4

Les dépendances ... 4

Classe principale ... 4

Fichier de configuration de l'application ... 5

Déclaration de haricot par annotation ... 5

Déclaration de bean par configuration d'application ... 6

Constructeur Injection ... 6

Injection de propriété ... 6

Crochets PostConstruct / PreDestroy ... 7

Qu'est-ce que Spring Framework, pourquoi devrions-nous y aller? ... 7

Alors, pourquoi choisir le printemps comme struts est là ... 8

Chapitre 2: Base de printemps ... 10

Examples ... 10

Introduction à Spring Core ... 10

Comprendre comment Spring gère les dépendances? ... 11

Chapitre 3: Comprendre le dispatcher-servlet.xml ... 15

Introduction ... 15

Examples ... 15

dispatcher-servlet.xml ... 15

Configuration du servlet du répartiteur dans web.xml ... 16

Chapitre 4: Configuration ApplicationContext ... 17

Remarques ... 17

Examples ... 17

Configuration Java 17

Configuration XML 19

Autowiring 20

Amorcer l'applicationContext 21

Configuration Java 21

Xml Config 21

Autowiring 22

Chapitre 5: Créer et utiliser des haricots 23

Examples 23

En utilisant tous les haricots d'un type spécifique 23

Déclarer le haricot 24

Annotation de base automatique 25

Utilisation de FactoryBean pour l'instanciation dynamique du bean 26

Injecter des haricots à prototypes dans des singletons 27

En utilisant des instances de bean spécifiques avec @Qualifier 30

En utilisant des instances spécifiques de classes en utilisant des paramètres de type géné 32

Chapitre 6: Enregistrement de haricot conditionnel au printemps 34

Remarques 34

Examples 34

Enregistrer les haricots uniquement lorsqu'une propriété ou une valeur est spécifiée 34

Annotations de condition 35

Conditions de classe 35

Conditions de haricots 35

Conditions de propriété 35

Conditions de ressources 36

Conditions d'application Web 36

Conditions d'expression de SpEL 36

Chapitre 7: Étendues de haricots 37

Examples 37

Portée singleton 37

Haricots singleton paresseux 38

Portée du prototype ... 38
Étendues supplémentaires dans des contextes Web ... 40
Configuration XML ... 40
Configuration Java (avant le printemps 4.3) ... 40
Configuration Java (après Spring 4.3) ... 41
Composants pilotés par annotation ... 41
**Chapitre 8: Exécution des tâches et planification ... 43**
Examples ... 43
Activer la planification ... 43
Délai fixe ... 43
Taux fixe ... 43
Expression cron ... 43
**Chapitre 9: Initialisation paresseuse du printemps ... 47**
Examples ... 47
Initialisation différée dans la classe de configuration ... 47
Pour la numérisation de composants et le câblage automatique ... 47
Exemple d'initiation paresseuse au printemps ... 47
**Chapitre 10: Injection de dépendance (DI) et inversion de contrôle (IoC) ... 49**
Remarques ... 49
Examples ... 50
Injection manuelle d'une dépendance via la configuration XML ... 50
Injection manuelle d'une dépendance via la configuration Java ... 51
En utilisant une dépendance via la configuration XML ... 52
En utilisant une dépendance via la configuration Java ... 53
**Chapitre 11: JdbcTemplate ... 55**
Introduction ... 55
Examples ... 55
Méthodes de requête de base ... 55
Requête pour la liste de cartes ... 55
SQLRowSet ... 56
Opérations par lots ... 56
Extension NamedParameterJdbcTemplate de JdbcTemplate ... 57

Chapitre 12: Langage d'expression du printemps (SpEL) 59
Examples 59
Référence de syntaxe 59
Expressions littérales 59
Liste en ligne 59
Cartes en ligne 59
Méthodes d'invocation 59
Chapitre 13: Obtenir un SqlRowSet à partir de SimpleJdbcCall 60
Introduction 60
Examples 60
Création de SimpleJdbcCall 60
Bases de données Oracle 61
Chapitre 14: Profil de printemps 64
Examples 64
Les profils de printemps permettent de configurer les pièces disponibles pour certains env 64
Chapitre 15: RestTemplate 65
Examples 65
Téléchargement d'un fichier volumineux 65
Utilisation de l'authentification de base préemptive avec RestTemplate et HttpClient 65
Utilisation de l'authentification de base avec HttpClient de HttpComponent 67
Définition d'en-têtes sur la demande Spring RestTemplate 67
Résultats génériques de Spring RestTemplate 68
Chapitre 16: SOAP WS Consommation 69
Examples 69
Consommer un WS SOAP avec l'authentification de base 69
Chapitre 17: Source de propriété 70
Examples 70
Annotation 70
Exemple de configuration XML à l'aide de PropertyPlaceholderConfigurer 70
Chapitre 18: Validation du haricot JSR 303 au printemps 72
Introduction 72

Examples ... 72
JSR303 Validation basée sur des annotations dans des exemples de ressorts ... 72
Spring JSR 303 Validation - Personnalisez les messages d'erreur ... 74
Utilisation de @Valid pour valider les POJO imbriqués ... 76
**Crédits ... 78**

# À propos

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.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

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

# Chapitre 1: Commencer avec le printemps

## Remarques

Spring Framework est un framework d'application open source et d'inversion de conteneur de contrôle pour la plate-forme Java.

## Versions

| Version | Date de sortie |
|---|---|
| 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

### Configuration (configuration XML)

Étapes pour créer Hello Spring:

0. Examinez Spring Boot pour voir si cela répondrait mieux à vos besoins.
1. Avoir un projet configuré avec les dépendances correctes. Il est recommandé d'utiliser Maven ou Gradle .
2. créer une classe POJO, par exemple `Employee.java`
3. créer un fichier XML dans lequel vous pouvez définir votre classe et vos variables. par

exemple `beans.xml`

4. Créez votre classe principale, par exemple `Customer.java`
5. Inclure les Spring-Beans (et ses dépendances transitives!) En tant que dépendance.

`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();
   }
}
```

## Présentation des fonctionnalités de base du ressort par exemple

## La description

Voici un exemple d'exécution autonome incluant / showcasing: *dépendances* minimales requises, *configuration* Java, *déclaration Bean* par annotation et configuration Java, *injection de dépendance* par constructeur et par propriété, et hooks *Pre / Post* .

## Les dépendances

Ces dépendances sont nécessaires dans le classpath:

1. noyau de printemps
2. contexte de printemps
3. haricots de printemps
4. printemps-aop
5. expression de printemps
6. commons-journalisation

---

## Classe principale

À partir de la fin, il s'agit de notre classe principale qui sert d'espace réservé pour la méthode `main()` qui initialise le contexte d'application en pointant sur la classe Configuration et charge tous les différents beans nécessaires pour présenter des fonctionnalités particulières.

```
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();
    }
}
```

---

# Fichier de configuration de l'application

La classe de configuration est annotée par `@Configuration` et est utilisée comme paramètre dans le contexte d'application initialisé. L'annotation `@ComponentScan` au niveau de la classe de la classe de configuration pointe vers un package à analyser pour les beans et les dépendances enregistrés à l'aide d'annotations. Enfin, l'annotation `@Bean` sert de définition de bean dans la classe de configuration.

```
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();
    }
}
```

---

# Déclaration de haricot par annotation

L'annotation `@Component` sert à délimiter le POJO en tant que bean Spring disponible pour l'enregistrement lors de l'analyse des composants.

```
@Component
public class BeanDeclaredByAnnotation {

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

---

## Déclaration de bean par configuration d'application

Notez que nous n'avons pas besoin d'annoter ou de marquer notre POJO, car la déclaration / définition du bean se produit dans le fichier de classe Configuration de l'application.

```
public class BeanDeclaredInAppConfig {

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

---

## Constructeur Injection

Notez que l'annotation `@Autowired` est définie au niveau du constructeur. Notez également que, sauf définition explicite par nom, le démarrage automatique par défaut se produit en *fonction du type* du bean (dans cet exemple, `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();
    }
}
```

---

## Injection de propriété

Notez que l'annotation `@Autowired` délimite la méthode setter dont le nom suit le standard 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 !");
    }
}
```

---

# Crochets PostConstruct / PreDestroy

Nous pouvons intercepter l'initialisation et la destruction d'un bean par les `@PostConstruct` et `@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");
    }
}
```

## Qu'est-ce que Spring Framework, pourquoi devrions-nous y aller?

Spring est un framework qui fournit un tas de classes. En utilisant ceci, nous n'avons pas besoin d'écrire la logique de la plaque chauffante dans notre code, donc Spring fournit une couche abstraite sur J2ee.

---

Par exemple, dans Simple JDBC Application, le programmeur est responsable de

1. Chargement de la classe de pilote
2. Créer la connexion
3. Création d'un objet d'instruction
4. Traitement des exceptions
5. Créer une requête
6. Exécution d'une requête
7. Fermer la connexion

Ce qui est traité comme un code standard comme chaque programmeur écrit le même code. Donc, pour simplifier, le cadre prend en charge la logique standard et le programmeur doit écrire uniquement la logique métier. Ainsi, en utilisant le framework Spring, nous pouvons développer rapidement des projets avec un minimum de lignes de code, sans aucun bogue, le coût de développement et le temps également réduit.

## Alors, pourquoi choisir le printemps comme struts est là

Strut est un cadre qui fournit une solution aux aspects Web uniquement et les structures sont invasives dans la nature. Le printemps a de nombreuses fonctionnalités par rapport aux jambes de force, nous devons donc choisir le printemps.

1. Spring est de nature ***non invasive*** : cela signifie que vous n'avez pas besoin d'étendre de classes ou d'implémenter des interfaces dans votre classe.
2. Le printemps est ***polyvalent*** : cela signifie qu'il peut être intégré à toute technologie existante dans votre projet.
3. Spring fournit un développement de projet de ***bout en bout*** : cela signifie que nous pouvons développer tous les modules tels que la couche métier, la couche de persistance.
4. Le printemps est ***léger*** : cela signifie que si vous souhaitez travailler sur un module particulier, vous n'avez pas besoin d'apprendre le printemps complet, mais seulement d'apprendre ce module particulier (par exemple, Spring Jdbc, Spring DAO)
5. Spring prend en ***charge l'injection de dépendance*** .
6. Spring prend ***en*** charge le ***développement de plusieurs projets,*** par exemple: Application Core java, application Web, application distribuée, application d'entreprise.
7. Spring soutient la programmation orientée aspect pour des problèmes transversaux.

On peut donc enfin dire que Spring est une alternative aux Struts. Mais Spring ne remplace pas l'API J2EE, car les classes fournies par Spring utilisent en interne des classes d'API J2EE. Le printemps est un vaste cadre qui a été divisé en plusieurs modules. Aucun module n'est dépendant d'un autre, à l'exception de Spring Core. Certains modules importants sont

1. Base de printemps
2. JDBC de printemps
3. AOP de printemps
4. Transaction de printemps
5. ORM de printemps
6. MVC de printemps

Lire Commencer avec le printemps en ligne: https://riptutorial.com/fr/spring/topic/786/commencer-

avec-le-printemps

# Chapitre 2: Base de printemps

## Examples

### Introduction à Spring Core

Le printemps est un vaste cadre, de sorte que le cadre Spring a été divisé en plusieurs modules, ce qui rend le printemps léger. Certains modules importants sont:

1. Base de printemps
2. AOP de printemps
3. JDBC de printemps
4. Transaction de printemps
5. ORM de printemps
6. MVC de printemps

Tous les modules de Spring sont indépendants les uns des autres, à l'exception de Spring Core. Comme le noyau de base est le module de base, nous devons donc utiliser Spring Core dans tous les modules.

***Base de printemps***

Spring Core parle de la gestion des dépendances. Cela signifie que si des classes arbitraires sont fournies au printemps, Spring peut gérer les dépendances.

**Qu'est-ce qu'une dépendance?**
Du point de vue du projet, dans un projet ou une application, plusieurs classes ont des fonctionnalités différentes. et chaque classe nécessitait certaines fonctionnalités d'autres classes.

**Exemple:**

```
class Engine {

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

class Car {

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

Ici, la classe Engine est requise par la classe car nous pouvons donc dire que le moteur de classe dépend de la classe Car, donc au lieu de gérer ces dépendances par héritage ou de créer un objet comme jachère.

***Par héritage:***

```
class Engine {

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

class Car extends Engine {

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

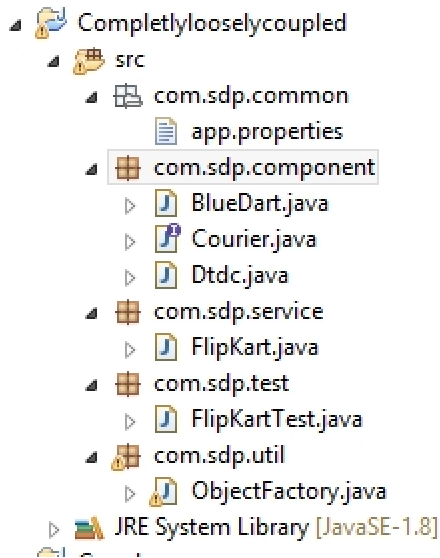
***En créant un objet de classe dépendante:***

```
class Engine {

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

class Car {

  Engine eng = new Engine();

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

Ainsi, au lieu de gérer la dépendance entre les classes, Spring core prend la responsabilité de la gestion des dépendances. Mais certaines règles existent, les classes doivent être conçues avec une technique de conception qui est le modèle de conception de la stratégie.

## Comprendre comment Spring gère les dépendances?

Permettez-moi d'écrire un morceau de code qui montre complètement couplé vaguement, alors vous pouvez facilement comprendre comment Spring core gère la dépendance en interne. Considérons un scénario, les affaires en ligne Flipkart est là, il utilise parfois DTDC ou service de messagerie Blue Dart, alors laissez-moi concevoir une application qui montre complètement couplé vaguement. Le répertoire Eclipse comme jachère:




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

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

}
```

// classes d'implémentation

```
  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";     }

}
```

// classe de composant

```
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);
    }

}
```

// Classes d'usine pour créer et renvoyer un objet

```
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;
}

    }
```

// fichier de propriétés

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

// classe de test

```
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 nous écrivons ce code, nous pouvons manuellement obtenir un couplage lâche, cela s’applique si toutes les classes veulent BlueDart ou Dtdc, mais si une classe veut BlueDart et une autre classe veut Dtdc, alors elle sera étroitement couplée. créer et gérer l'injection de dépendances Spring core prend la responsabilité de créer et de gérer les beans, Hope Cela vous sera utile, dans l'exemple suivant, nous verrons l'application! st sur Spring core avec des deitals

Lire Base de printemps en ligne: https://riptutorial.com/fr/spring/topic/7067/base-de-printemps

# Chapitre 3: Comprendre le dispatcher-servlet.xml

## Introduction

Dans Spring Web MVC, la classe DispatcherServlet fonctionne comme contrôleur frontal. Il est responsable de la gestion du flux de l'application Spring MVC.

DispatcherServlet est également comme le servlet normal doit être configuré dans web.xml

## Examples

### dispatcher-servlet.xml

C'est le fichier de configuration important où vous devez spécifier les composants ViewResolver et View.

L'élément context: component-scan définit le package de base où DispatcherServlet recherchera la classe du contrôleur.

Ici, la classe InternalResourceViewResolver est utilisée pour ViewResolver.

Le préfixe + chaîne renvoyé par la page du contrôleur + suffixe sera appelé pour le composant de vue.

Ce fichier xml doit être situé dans le répertoire 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>
```

### Configuration du servlet du répartiteur dans web.xml

Dans ce fichier XML, nous spécifions la classe de servlet DispatcherServlet qui agit en tant que contrôleur frontal dans Spring Web MVC. Toutes les demandes entrantes pour le fichier HTML seront transmises au 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>
```

Lire Comprendre le dispatcher-servlet.xml en ligne:
https://riptutorial.com/fr/spring/topic/10092/comprendre-le-dispatcher-servlet-xml

# Chapitre 4: Configuration ApplicationContext

## Remarques

Spring a fait en sorte que la configuration d'un `ApplicationContext` soit extrêmement flexible. Il existe de nombreuses manières d'appliquer chaque type de configuration, et elles peuvent toutes être combinées et harmonisées.

**La configuration Java** est une forme de configuration *explicite* . Une classe annotée `@Configuration` est utilisée pour spécifier les beans qui feront partie de `ApplicationContext` , ainsi que pour définir et câbler les dépendances de chaque bean.

**La configuration XML** est une forme de configuration *explicite* . Un schéma XML spécifique est utilisé pour définir les beans qui feront partie de `ApplicationContext` . Ce même schéma est utilisé pour définir et câbler les dépendances de chaque bean.

**La création automatique** est une forme de configuration *automatique* . Certaines annotations sont utilisées dans les définitions de classes pour déterminer quels beans feront partie de `ApplicationContext` , et d'autres annotations sont utilisées pour connecter les dépendances de ces beans.

## Examples

### Configuration Java

La configuration Java est généralement effectuée en appliquant l'annotation `@Configuration` à une classe pour suggérer qu'une classe contient des définitions de bean. Les définitions de bean sont spécifiées en appliquant l'annotation `@Bean` à une méthode qui renvoie un objet.

```
@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;
    }
}
```

## Configuration XML

La configuration XML est généralement effectuée en définissant des beans dans un fichier xml, en utilisant le schéma `beans` spécifique à Spring. Sous l'élément `beans` racines, la définition de haricot typique serait faite en utilisant le sous-élément `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;
    }
}
```

## Autowiring

L'enregistrement automatique est effectué à l'aide d'une annotation de *stéréotype* pour spécifier les classes qui seront des beans dans `ApplicationContext` , et à l'aide des annotations `Autowired` et `Value` pour spécifier les dépendances du bean. La caractéristique unique de la création

automatique est qu’il n’existe pas de définition externe d’ `ApplicationContext` , car elle se fait dans les classes qui sont les beans eux-mêmes.

```
@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; }
}
```

### Amorcer l'applicationContext

## Configuration Java

La classe de configuration doit uniquement être une classe figurant dans le chemin de classe de votre application et visible par la classe principale de vos applications.

```
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...
}
```

---

## Xml Config

Le fichier de configuration xml ne doit figurer que dans le classpath de votre application.

```
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>
```

---

# Autowiring

La création automatique doit savoir quels packages de base rechercher les beans annotés ( `@Component` ). Ceci est spécifié via la `#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...
}
```

Lire Configuration ApplicationContext en ligne: https://riptutorial.com/fr/spring/topic/3844/configuration-applicationcontext

# Chapitre 5: Créer et utiliser des haricots

## Examples

### En utilisant tous les haricots d'un type spécifique

Si vous disposez de plusieurs implémentations de la même interface, Spring peut les envoyer automatiquement dans un objet de collection. Je vais utiliser un exemple utilisant un modèle de validateur [1]

Classe Foo:

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

Interface:

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

Classe de validateur de nom:

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

Classe de validation de courrier électronique:

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

Vous pouvez maintenant envoyer ces validateurs individuellement ou ensemble dans une classe.

Interface:

```
public interface FooService {
```

```
    public void handleFoo(Foo foo);
}
```

Classe:

```
@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);
       }
    }
}
```

Il est intéressant de noter que si vous avez plus d'une implémentation d'une interface dans le conteneur Spring IoC et que vous ne spécifiez pas celle que vous souhaitez utiliser avec l'annotation `@Qualifier` , Spring lancera une exception en essayant de démarrer, car elle a gagné. t savoir quelle instance utiliser.

1: Ce n'est pas la bonne façon de faire de telles validations simples. Ceci est un exemple simple d'auto-activation. Si vous voulez une idée d'une méthode de validation beaucoup plus facile, regardez comment Spring effectue la validation avec les annotations.

## Déclarer le haricot

Pour déclarer un bean, `@Bean` simplement une méthode avec l'annotation `@Bean` ou `@Bean` une classe avec l'annotation `@Component` (les annotations `@Service` , `@Repository` , `@Controller` peuvent également être utilisées).

Lorsque JavaConfig rencontre une telle méthode, il exécute cette méthode et enregistre la valeur de retour en tant que bean dans une BeanFactory. Par défaut, le nom du bean sera celui du nom de la méthode.

Nous pouvons créer un haricot de trois manières différentes:

1. **Utilisation de la configuration basée sur Java** : Dans le fichier de configuration, nous devons déclarer le bean en utilisant l'annotation @bean

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

2. **Utilisation de la configuration basée sur XML** : Pour une configuration basée sur XML,

nous devons créer declare bean dans le XML de configuration de l'application, par exemple

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

3. **Composant piloté** par des annotations: Pour les composants pilotés par des annotations, nous devons ajouter l'annotation @Component à la classe que nous voulons déclarer en tant que bean.

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

Les trois beans `BeanFactory` `transferService` sont `BeanFactory` disponibles dans `BeanFactory` ou `ApplicationContext` .

## Annotation de base automatique

Interface:

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

Classe:

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

Il convient de noter qu'une classe doit implémenter une interface pour que Spring puisse générer automatiquement cette classe. Il existe une méthode pour autoriser Spring à créer automatiquement des classes autonomes en utilisant le tissage du temps de chargement, mais cela est hors de portée pour cet exemple.

Vous pouvez accéder à ce bean dans toute classe instanciée par le conteneur Spring IoC à l'aide de l'annotation `@Autowired` .

Usage:

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

L'annotation `@Autowired` abord de s'exprimer automatiquement par type, puis utilisera le nom du

bean en cas d'ambiguïté.

Cette annotation peut être appliquée de différentes manières.

Injection constructeur:

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

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

Injection de champ:

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

Injection Setter:

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

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

## Utilisation de FactoryBean pour l'instanciation dynamique du bean

Afin de décider dynamiquement des beans à injecter, nous pouvons utiliser `FactoryBean` . Ce sont des classes qui implémentent le modèle de méthode de fabrique, fournissant des instances de beans pour le conteneur. Ils sont reconnus par le printemps et peuvent être utilisés de manière transparente, sans qu'il soit nécessaire de savoir que le grain provient d'une usine. Par exemple:

```
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";
```

```
    }
}
```

Configuration:

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

Obtenir le haricot:

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

Pour obtenir le FactoryBean réel, utilisez le préfixe ampersand avant le nom du bean:

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

S'il vous plaît noter que vous ne pouvez utiliser des `prototype` ou `singleton` champs - pour changer la portée de `prototype` override `isSingleton` méthode:

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

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

Notez que la portée fait référence aux instances réelles en cours de création, pas au bean de fabrication lui-même.

## Injecter des haricots à prototypes dans des singletons

Le conteneur crée un bean singleton et n'y injecte qu'une seule fois des collaborateurs. Ce comportement n'est pas souhaitable lorsque le bean singleton a un collaborateur de portée prototype, car le bean à portée de prototype doit être injecté à chaque fois qu'il est accédé via l'accesseur.

Il existe plusieurs solutions à ce problème:

1. Utiliser l'injection de la méthode de recherche
2. Récupère un bean prototype sur `javax.inject.Provider`
3. Récupérer un bean prototype à portée via `org.springframework.beans.factory.ObjectFactory` (équivalent à # 2, mais avec la classe spécifique à Spring)

4. Rendre conscient un conteneur de beans singleton via l'implémentation de l'interface `ApplicationContextAware`

Les approches n ° 3 et n ° 4 sont généralement déconseillées car elles associent fortement une application au framework Spring. Ainsi, ils ne sont pas couverts dans cet exemple.

**Injection de méthode de recherche via une configuration XML et une méthode abstraite**

Classes 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>
```

**Injection de la méthode de recherche via la configuration Java et @Component**

Classes 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();
    }
}
```

Configuration 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();
    }
}
```

### Injection de méthode de recherche manuelle via la configuration Java

Classes 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
}
```

Configuration 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();
            }
        };
    }
}
```

### Injection d'un bean à protoype dans singleton via `javax.inject.Provider`

Classes 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>
```

Les mêmes approches peuvent également être utilisées pour d'autres domaines (par exemple, pour l'injection d'un bean à portée de requête dans singleton).

## En utilisant des instances de bean spécifiques avec @Qualifier

Si vous disposez de plusieurs implémentations de la même interface, Spring doit savoir laquelle elle doit utiliser dans une classe. Je vais utiliser un modèle de validateur dans cet exemple. [1]

Classe Foo:

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

Interface:

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

Classe de validateur de nom:

```
@Component(value="FooNameValidator")
public class FooNameValidator implements FooValidator {
```

```
    @Override
    public Foo validate(Foo foo) {
        //Validation logic goes here.
    }
}
```

Classe de validation de courrier électronique:

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

Vous pouvez maintenant envoyer ces validateurs individuellement dans une classe.

Interface:

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

Classe:

```
@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);
    }
}
```

Il est intéressant de noter que si vous avez plus d'une implémentation d'une interface dans le conteneur Spring IoC et que vous ne spécifiez pas celle que vous souhaitez utiliser avec l'annotation `@Qualifier` , Spring lancera une exception en essayant de démarrer, car elle a gagné. t savoir quelle instance utiliser.

1: Ce n'est pas la bonne façon de faire de telles validations simples. Ceci est un exemple simple d'auto-activation. Si

vous voulez une idée d'une méthode de validation beaucoup plus facile, regardez comment Spring effectue la validation avec les annotations.

## En utilisant des instances spécifiques de classes en utilisant des paramètres de type génériques

Si vous disposez d'une interface avec un paramètre de type générique, Spring peut l'utiliser pour ne générer que des implémentations implémentant un paramètre de type que vous spécifiez.

Interface:

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

Foo Validator Classe:

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

Classe de validateur de barre:

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

Vous pouvez maintenant envoyer ces validateurs en utilisant les paramètres de type pour décider quelle instance envoyer automatiquement.

Interface:

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

Classe:

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

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

Lire Créer et utiliser des haricots en ligne: https://riptutorial.com/fr/spring/topic/3182/creer-et-utiliser-des-haricots

# Chapitre 6: Enregistrement de haricot conditionnel au printemps

## Remarques

Point important à noter lors de l'utilisation de la condition

- La classe de condition est appelée classe directe (pas comme haricot printanier), elle **ne peut donc pas** utiliser l'injection de propriété `@Value` , c'est-à-dire qu'aucun autre haricot de printemps ne peut y être injecté.
- À partir de documents Java - Les *conditions doivent respecter les mêmes restrictions que* `BeanFactoryPostProcessor` *et veiller à ne jamais interagir avec les instances du bean* . Les restrictions mentionnées ici sont les suivantes: *Un* `BeanFactoryPostProcessor` *peut interagir avec et modifier les définitions de bean, mais jamais les instances de bean. Cela pourrait provoquer une instanciation prématurée du haricot, violer le contenant et entraîner des effets secondaires imprévus.*

## Examples

### Enregistrer les haricots uniquement lorsqu'une propriété ou une valeur est spécifiée

Un haricot de printemps peut être configuré de manière à ce qu'il ne soit enregistré *que* s'il possède une valeur particulière *ou si* une propriété spécifiée est remplie. Pour ce faire, implémentez `Condition.matches` pour vérifier la propriété / valeur:

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

Dans la configuration Java, utilisez l'implémentation ci-dessus comme condition pour enregistrer le bean. Notez l'utilisation de l'annotation @Conditional .

```
@Configuration
public class MyAppConfig {

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

Dans `PropertyCondition` , vous pouvez évaluer n'importe quel nombre de conditions. Cependant, il est conseillé de séparer la mise en œuvre pour chaque condition afin de ne pas les coupler librement. Par exemple:

```
@Configuration
public class MyAppConfig {

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

### Annotations de condition

À l' `@conditional` annotation `@conditional` principale, des annotations similaires peuvent être utilisées pour différents cas.

# Conditions de classe

Les annotations `@ConditionalOnClass` et `@ConditionalOnMissingClass` permettent d'inclure la configuration en fonction de la présence ou de l'absence de classes spécifiques.

Par exemple, lorsque `OObjectDatabaseTx.class` est ajouté aux dépendances et qu'il n'y a pas de bean `OrientWebConfigurer` nous créons le configurateur.

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

# Conditions de haricots

Les annotations `@ConditionalOnBean` et `@ConditionalOnMissingBean` permettent d' `@ConditionalOnMissingBean` un bean en fonction de la présence ou de l'absence de beans spécifiques. Vous pouvez utiliser l'attribut value pour spécifier les beans par type ou name pour spécifier les beans par nom. L'attribut de recherche vous permet de limiter la hiérarchie `ApplicationContext` prendre en compte lors de la recherche de beans.

Voir l'exemple ci-dessus lorsque nous vérifions s'il n'y a pas de bean défini.

# Conditions de propriété

L'annotation `@ConditionalOnProperty` permet d'inclure la configuration en fonction d'une propriété Spring Environment. Utilisez les attributs prefix et name pour spécifier la propriété à vérifier. Par défaut, toute propriété qui existe et n'est pas égale à `false` sera mise en correspondance. Vous pouvez également créer des vérifications plus avancées à l'aide des `havingValue` et `matchIfMissing` .

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

# Conditions de ressources

L'annotation `@ConditionalOnResource` permet d'inclure la configuration uniquement lorsqu'une ressource spécifique est présente.

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

# Conditions d'application Web

Les annotations `@ConditionalOnWebApplication` et `@ConditionalOnNotWebApplication` permettent d'inclure la configuration selon que l'application est une "application Web".

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

# Conditions d'expression de SpEL

L'annotation `@ConditionalOnExpression` permet d'inclure la configuration en fonction du résultat d'une expression SpEL.

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

Lire Enregistrement de haricot conditionnel au printemps en ligne: https://riptutorial.com/fr/spring/topic/4732/enregistrement-de-haricot-conditionnel-au-printemps

# Chapitre 7: Étendues de haricots

## Examples

### Portée singleton

Si un bean est défini avec la portée singleton, une seule instance d'objet unique sera initialisée dans le conteneur Spring. Toutes les demandes adressées à ce bean renverront la même instance partagée. C'est la portée par défaut lors de la définition d'un bean.

Compte tenu de la classe MyBean suivante:

```
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;
    }
}
```

On peut définir un bean singleton avec l'annotation @Bean:

```
@Configuration
public class SingletonConfiguration {

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

L'exemple suivant extrait deux fois le même bean du contexte Spring:

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

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

Lors de la connexion de la propriété singletonBean2, le message *"Propriété modifiée"* sera affiché, car nous venons de récupérer la même instance partagée.

Étant donné que l'instance est partagée entre différents composants, il est recommandé de définir la portée singleton pour les objets sans état.

# Haricots singleton paresseux

Par défaut, les beans singleton sont pré-instanciés. Par conséquent, l'instance d'objet partagé sera créée lors de la création du conteneur Spring. Si nous démarrons l'application, le message *"Initializing singleton bean ..."* sera affiché.

Si nous ne voulons pas que le bean soit pré-instancié, nous pouvons ajouter l'annotation @Lazy à la définition du bean. Cela empêchera le haricot d'être créé jusqu'à ce qu'il soit demandé pour la première fois.

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

Maintenant, si nous démarrons le conteneur Spring, aucun message *"Initializing lazy singleton bean ..."* ne s'affichera. Le haricot ne sera pas créé tant qu'il n'aura pas été demandé pour la première fois:

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

Si nous exécutons l'application avec les beans singleton et lazy singleton définis, elle produira les messages suivants:

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

### Portée du prototype

Un haricot prototype n'a pas été créé au démarrage du conteneur Spring. Au lieu de cela, une nouvelle instance sera créée à chaque fois qu'une demande de récupération de ce bean sera envoyée au conteneur. Cette étendue est recommandée pour les objets avec état, car son état ne sera pas partagé par les autres composants.

Pour définir un bean prototype, nous devons ajouter l'annotation @Scope en spécifiant le type de portée que nous voulons.

Compte tenu de la classe MyBean suivante:

```
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;
    }
}
```

Nous définissons une définition de haricot, indiquant sa portée en tant que prototype:

```
@Configuration
public class PrototypeConfiguration {

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

Pour voir comment cela fonctionne, nous récupérons le bean du conteneur Spring et définissons une valeur différente pour son champ de propriété. Ensuite, nous allons récupérer le bean du conteneur et rechercher sa valeur:

```
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());
```

En regardant le résultat suivant, nous pouvons voir comment une nouvelle instance a été créée sur chaque requête de bean:

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

Une erreur commune est de supposer que le bean est recréé par invocation ou par thread, ce n'est **PAS** le cas. Au lieu de cela, une instance est créée PER INJECTION (ou récupération à partir du contexte). Si un haricot de portée Prototype n'est injecté que dans un seul haricot singleton, il n'y aura qu'une seule instance de ce haricot de portée Prototype.

Spring ne gère pas le cycle de vie complet d'un bean prototype: le conteneur instancie, configure, décore et assemble un objet prototype, le transmet au client et n'a plus aucune connaissance de cette instance de prototype.

### Étendues supplémentaires dans des contextes Web

Plusieurs portées sont disponibles uniquement dans un contexte d'application Web:

- **request** - Une nouvelle instance de bean est créée par requête HTTP
- **session** - une nouvelle instance de bean est créée par session HTTP
- **application** - une nouvelle instance de bean est créée par `ServletContext`
- **globalSession** - Une nouvelle instance de bean est créée par session globale dans un environnement de portlet (dans l'environnement de servlet, la portée de session globale est égale à la portée de la session)
- **websocket** - Une nouvelle instance de bean est créée par session WebSocket

Aucune configuration supplémentaire n'est requise pour déclarer et accéder à des beans de portée Web dans l'environnement Spring Web MVC.

## Configuration 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"/>
```

## Configuration Java (avant le printemps 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();
```

```
    }
}
```

## Configuration Java (après Spring 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();
    }
}
```

## Composants pilotés par annotation

```
@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 {
    ...
}
```

Lire Étendues de haricots en ligne: https://riptutorial.com/fr/spring/topic/3526/etendues-de-haricots

# Chapitre 8: Exécution des tâches et planification

## Examples

### Activer la planification

Spring fournit un support utile pour la planification des tâches. Pour l'activer, `@Configuration` simplement l'une de vos classes `@EnableScheduling` avec `@EnableScheduling` :

```
@Configuration
@EnableScheduling
public class MyConfig {

    // Here it goes your configuration
}
```

### Délai fixe

Si nous voulons que du code soit exécuté périodiquement après l'exécution qui était avant, nous devrions utiliser un délai fixe (mesuré en millisecondes):

```
@Component
public class MyScheduler{

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

### Taux fixe

Si nous voulons que quelque chose soit exécuté périodiquement, ce code sera déclenché une fois par la valeur en millisecondes que nous spécifions:

```
@Component
public class MyScheduler{

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

### Expression cron

Une expression Cron se compose de six champs séquentiels -

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

et est déclaré comme suit

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

Nous pouvons également définir le fuseau horaire comme -

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

**Remarques: -**

```
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
```

**Exemple: -**

```
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
```

Une méthode déclarée avec `@Scheduled()` est appelée explicitement pour chaque cas correspondant.

Si nous voulons qu'un code soit exécuté lorsqu'une expression cron est rencontrée, nous devons le spécifier dans l'annotation:

```
@Component
public class MyScheduler{

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

Si nous voulons imprimer l'heure actuelle dans notre console après 5 secondes -

```
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()));
    }

}
```

### Exemple utilisant la configuration XML:

Exemple de classe:

```
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()));
    }

}
```

Exemple 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>
```

Lire Exécution des tâches et planification en ligne:
https://riptutorial.com/fr/spring/topic/6080/execution-des-taches-et-planification

# Chapitre 9: Initialisation paresseuse du printemps

## Examples

### Initialisation différée dans la classe de configuration

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

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

### Pour la numérisation de composants et le câblage automatique

```
@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;

    .......
 }
```

### Exemple d'initiation paresseuse au printemps

Le `@Lazy` nous permet de demander au conteneur IOC de retarder l'initialisation d'un bean. Par défaut, les beans sont instanciés dès que le conteneur IOC est créé. Les `@Lazy` nous permettent de modifier ce processus d'instanciation.

lazy-init au printemps est l'attribut de la balise bean. Les valeurs de lazy-init sont vraies et fausses. Si lazy-init a la valeur true, alors ce bean sera initialisé lorsqu'une requête est envoyée au bean. Ce haricot ne sera pas initialisé lorsque le conteneur de printemps est initialisé. Si lazy-init est faux, le bean sera initialisé avec l'initialisation du conteneur de printemps et c'est le comportement par défaut.

### 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");
   }
}
```

### SpringTest.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");
   }
}
```

### Sortie

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

Lire Initialisation paresseuse du printemps en ligne:
https://riptutorial.com/fr/spring/topic/6221/initialisation-paresseuse-du-printemps

# Chapitre 10: Injection de dépendance (DI) et inversion de contrôle (IoC)

## Remarques

Le code source des grandes applications logicielles est généralement organisé en plusieurs unités. La définition d'une unité varie normalement selon le langage de programmation utilisé. Par exemple, le code écrit dans un langage de programmation procédural (comme C) est organisé en `functions` ou `procedures` . De même, le code dans un langage de programmation orienté objet (comme Java, Scala et C #) est organisé en `classes` , `interfaces` , etc. Ces unités d'organisation de code peuvent être considérées comme des unités individuelles constituant l'application logicielle globale.

Lorsque les applications ont plusieurs unités, les interdépendances entre ces unités surviennent lorsqu'une unité doit utiliser d'autres pour compléter ses fonctionnalités. Les unités dépendantes peuvent être considérées comme des `consumers` et des unités sur lesquelles elles dépendent en tant que `providers` de fonctionnalités spécifiques.

L'approche de programmation la plus simple consiste pour les consommateurs à contrôler entièrement le flux d'une application logicielle en décidant quels fournisseurs doivent être instanciés, utilisés et détruits à quels moments de l'exécution globale de l'application. On dit que les consommateurs ont un contrôle total sur les fournisseurs pendant le flux d'exécution, qui sont des `dependencies` pour les consommateurs. Dans le cas où les fournisseurs ont leurs propres dépendances, les consommateurs peuvent avoir à s'inquiéter de la manière dont les fournisseurs doivent être initialisés (et libérés), ce qui rend le flux de contrôle de plus en plus compliqué à mesure que le nombre d'unités du logiciel augmente. Cette approche augmente également le couplage entre les unités, rendant de plus en plus difficile le changement individuel des unités sans se soucier de casser les autres parties du logiciel.

Inversion de contrôle (IoC) est un principe de conception qui préconise l'externalisation des activités de flux de contrôle telles que la découverte, l'instanciation et la destruction des unités dans un cadre indépendant des consommateurs et des fournisseurs. Le principe sous-jacent de l’IoC est de découpler les consommateurs et les fournisseurs, ce qui évite aux unités logicielles de s’inquiéter de la découverte, de l’instanciation et du nettoyage de leurs dépendances, et permet aux unités de se concentrer sur leurs propres fonctionnalités. Ce découplage permet de conserver le logiciel extensible et maintenable.

L'injection de dépendance est l'une des techniques permettant d'implémenter le principe d'inversion de contrôle selon lequel des instances de dépendances (fournisseurs) sont injectées dans une unité logicielle (le consommateur) au lieu que le consommateur les trouve et les instancie.

Le framework Spring contient un module d'injection de dépendances qui permet d'injecter des beans gérés par Spring dans d'autres beans gérés par Spring en tant que dépendances.

# Examples

## Injection manuelle d'une dépendance via la configuration XML

Considérons les classes Java suivantes:

```
class Foo {
  private Bar bar;

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

Comme on peut le voir, la classe `Foo` a besoin d'appeler la méthode `baz` sur une instance d' une autre classe `Bar` pour sa méthode `foo` pour travailler avec succès. `Bar` est considéré comme une dépendance pour `Foo` car `Foo` ne peut pas fonctionner correctement sans une instance de `Bar` .

Injection de constructeur

Lors de l'utilisation de la configuration XML pour le framework Spring afin de définir des beans gérés par Spring, un bean de type `Foo` peut être configuré comme suit:

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

ou bien (plus verbeux):

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

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

Dans les deux cas, le framework Spring crée d'abord une instance de `Bar` et l' `injects` dans une instance de `Foo` . Cet exemple suppose que la classe `Foo` a un constructeur capable de prendre une instance de `Bar` comme paramètre, à savoir:

```
class Foo {
  private Bar bar;

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

Ce style est appelé **injection de constructeur** car la dépendance (instance `Bar` ) est injectée via le constructeur de classe.

Injection de propriété

Une autre option pour injecter la dépendance `Bar` dans `Foo` est:

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

ou bien (plus verbeux):

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

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

Cela nécessite que la classe `Foo` ait une méthode de réglage qui accepte une instance de `Bar` , telle que:

```
class Foo {
  private Bar bar;

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

## Injection manuelle d'une dépendance via la configuration Java

Les mêmes exemples que ceux illustrés ci-dessus avec la configuration XML peuvent être réécrits avec la configuration Java comme suit.

Injection de constructeur

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

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

Injection de propriété

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

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

```
    return foo;
  }
}
```

## En utilisant une dépendance via la configuration XML

Les dépendances peuvent être automatiquement générées lors de l'utilisation de la fonctionnalité d'analyse de composants du framework Spring. Pour que le démarrage automatique fonctionne, la configuration XML suivante doit être effectuée:

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

where, `base-package` est le package Java complet dans lequel Spring doit effectuer une analyse des composants.

Injection de constructeur

Les dépendances peuvent être injectées via le constructeur de classe comme suit:

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

@Component
class Foo {
  private Bar bar;

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

Ici, `@Autowired` est une annotation spécifique à Spring. Spring prend également en charge JSR-299 pour permettre la portabilité des applications vers d'autres structures d'injection de dépendances basées sur Java. Cela permet à `@Autowired` d'être remplacé par `@Inject` tant que:

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

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

Injection de propriété

Les dépendances peuvent également être injectées en utilisant les méthodes setter comme suit:

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

  @Autowired
```

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

Injection sur le terrain

Le démarrage automatique permet également d'initialiser les champs directement dans les instances de classe, comme suit:

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

Pour les versions de printemps 4.1 et ultérieures, vous pouvez utiliser Facultatif pour les dépendances facultatives.

```
@Component
class Foo {

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

La même approche peut être utilisée pour les constructeurs DI.

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

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

## En utilisant une dépendance via la configuration Java

L'injection de constructeur via la configuration Java peut également utiliser le processus de création automatique, tel que:

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

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

Lire Injection de dépendance (DI) et inversion de contrôle (IoC) en ligne: https://riptutorial.com/fr/spring/topic/7295/injection-de-dependance--di--et-inversion-de-controle--

ioc-

# Chapitre 11: JdbcTemplate

## Introduction

La classe JdbcTemplate exécute des requêtes SQL, des instructions de mise à jour et des appels de procédures stockées, effectue une itération sur ResultSets et extrait les valeurs de paramètre renvoyées. Il intercepte également les exceptions JDBC et les traduit en une hiérarchie d'exceptions générique, plus informative, définie dans le package org.springframework.dao.

Les instances de la classe JdbcTemplate sont threadsafe une fois configurées pour pouvoir injecter en toute sécurité cette référence partagée dans plusieurs DAO.

## Examples

### Méthodes de requête de base

Certaines des méthodes queryFor * disponibles dans JdbcTemplate sont utiles pour les instructions SQL simples exécutant des opérations CRUD.

**Demander la date**

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

**Interroger pour un entier**

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

OU

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

**Requête pour String**

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

**Requête pour la liste**

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

### Requête pour la liste de cartes

```
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("---”);
}
```

OU

```
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;
  }

});
```

## Opérations par lots

JdbcTemplate fournit également des méthodes pratiques pour exécuter des opérations par lots.

**Insert de lot**

```
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();
    }
});
```

**Mise à jour par lot**

```
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();
    }
});
```

Il existe d'autres méthodes batchUpdate qui acceptent List of object array comme paramètres d'entrée. Ces méthodes utilisent en interne BatchPreparedStatementSetter pour définir les valeurs de la liste des tableaux dans une instruction sql.

## Extension NamedParameterJdbcTemplate de JdbcTemplate

> La classe `NamedParameterJdbcTemplate` ajoute la prise en charge de la programmation d'instructions JDBC à l'aide de paramètres nommés, par opposition à la programmation d'instructions JDBC en utilisant uniquement des arguments classiques ('?'). La classe `NamedParameterJdbcTemplate JdbcTemplate` un `JdbcTemplate` et délègue au `JdbcTemplate` une grande partie de son travail.

```
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);
```

Lire JdbcTemplate en ligne: https://riptutorial.com/fr/spring/topic/7742/jdbctemplate

# Chapitre 12: Langage d'expression du printemps (SpEL)

## Examples

### Référence de syntaxe

Vous pouvez utiliser `@Value("#{expression}")` pour injecter une valeur à l'exécution, dans laquelle l' `expression` est une expression SpEL.

## Expressions littérales

Les types pris en charge incluent les chaînes, les dates, les valeurs numériques (int, réel et hexadécimal), booléennes et nulles.

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

## Liste en ligne

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

## Cartes en ligne

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

## Méthodes d'invocation

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

Lire Langage d'expression du printemps (SpEL) en ligne:
https://riptutorial.com/fr/spring/topic/8109/langage-d-expression-du-printemps--spel-

# Chapitre 13: Obtenir un SqlRowSet à partir de SimpleJdbcCall

## Introduction

Cela décrit comment obtenir directement un **SqlRowSet en** utilisant **SimpleJdbcCall** avec une procédure stockée dans votre base de données qui a un **paramètre de sortie du curseur** ,

Je travaille avec une base de données Oracle, j'ai essayé de créer un exemple qui devrait fonctionner pour d'autres bases de données, mon exemple Oracle détaille les problèmes avec Oracle.

## Examples

### Création de SimpleJdbcCall

En règle générale, vous souhaiterez créer vos SimpleJdbcCalls dans un service.

Cet exemple suppose que votre procédure a un seul paramètre de sortie qui est un curseur; vous devrez ajuster vos declareParameters pour correspondre à votre procédure.

```
@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");
    }
```

```
}
```

Il y a beaucoup d'options que vous pouvez utiliser ici:

- **withoutProcedureColumnMetaDataAccess () est** nécessaire si vous avez surchargé les noms de procédures ou si vous ne souhaitez pas que SimpleJdbcCall valide la base de données.
- **withReturnValue ()** si la procédure a une valeur de retour. La première valeur donnée à declareParameters définit la valeur de retour. De plus, si votre procédure est une fonction, utilisez **withFunctionName** et **executeFunction** lors de l'exécution.
- **withNamedBinding ()** si vous voulez donner des arguments en utilisant des noms au lieu de position.
- **useInParameterNames ()** définit l'ordre des arguments. Je pense que cela peut être nécessaire si vous transmettez vos arguments sous forme de liste au lieu d'une carte de nom d'argument à valoriser. Bien que cela ne soit nécessaire que si vous utilisez withoutProcedureColumnMetaDataAccess ()

## Bases de données Oracle

Il existe un certain nombre de problèmes avec Oracle. Voici comment les résoudre.

En supposant que votre paramètre de sortie de procédure soit le `ref cursor` , vous obtiendrez cette exception.

> java.sql.SQLException: Type de colonne non valide: 2012

Changez donc `Types.REF_CURSOR` à `OracleTypes.CURSOR` dans **simpleJdbcCall.declareParameters ()**

---

**Prise en charge d'OracleTypes**

*Vous n'avez peut-être besoin de le faire que si vous avez certains types de colonnes dans vos données.*

Le problème suivant est que les types propriétaires tels que `oracle.sql.TIMESTAMPTZ` provoqué cette erreur dans SqlRowSetResultSetExtractor:

> Type SQL non valide pour la colonne; l'exception imbriquée est java.sql.SQLException: type SQL non valide pour la colonne

Nous devons donc créer un **ResultSetExtractor** prenant en charge les types Oracle.
*Je vais expliquer la raison du mot de passe après ce code.*

```
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);
    }

}
```

Certains types Oracle nécessitent une connexion pour obtenir la valeur de colonne à partir d'un ResultSet. TIMESTAMPTZ est l'un de ces types. Ainsi, lorsque `rowSet.getTimestamp(colIndex)` est appelé, vous obtiendrez cette exception:

> Causée par: java.sql.SQLException: une ou plusieurs propriétés RowSet d'authentification non définies sur oracle.jdbc.rowset.OracleCachedRowSet.getConnectionInternal (OracleCachedRowSet.java:560) sur oracle.jdbc.rowset.OracleCachedRowSet.getTimestamp (OracleCachedRowSet.java : 3717) at org.springframework.jdbc.support.rowset.ResultSetWrappingSqlRowSet.getTimestamp

Si vous creusez dans ce code, vous verrez que OracleCachedRowSet a besoin du mot de passe ou d'un nom de source de données JNDI pour obtenir une connexion. Si vous préférez la recherche JNDI, vérifiez simplement que OracleCachedRowSet a défini DataSourceName.

Donc, dans mon service, j'invoque le mot de passe et déclare le paramètre de sortie comme suit:

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

Lire Obtenir un SqlRowSet à partir de SimpleJdbcCall en ligne:
https://riptutorial.com/fr/spring/topic/9235/obtenir-un-sqlrowset-a-partir-de-simplejdbccall

# Chapitre 14: Profil de printemps

## Examples

### Les profils de printemps permettent de configurer les pièces disponibles pour certains environnements

Tout `@Component` ou `@Configuration` pourrait être marqué avec l'annotation `@Profile`

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

    // ...
}
```

La même chose dans la configuration XML

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

Les profils actifs peuvent être configurés dans le fichier `application.properties`

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

ou spécifié depuis la ligne de commande

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

ou dans SpringBoot

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

Il est possible d'activer les profils dans Tests en utilisant l'annotation `@ActiveProfiles("dev")`

Lire Profil de printemps en ligne: https://riptutorial.com/fr/spring/topic/9981/profil-de-printemps

# Chapitre 15: RestTemplate

## Examples

### Téléchargement d'un fichier volumineux

Les méthodes `getForObject` et `getForEntity` de `RestTemplate` chargent la totalité de la réponse en mémoire. Cela ne convient pas au téléchargement de fichiers volumineux, car cela peut entraîner des exceptions de mémoire insuffisante. Cet exemple montre comment diffuser la réponse à une requête 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);
```

Notez que vous ne pouvez pas simplement renvoyer le `InputStream` partir de l'extracteur, car la méthode et le flux sous-jacents sont déjà fermés au retour de la méthode d'exécution.

### Utilisation de l'authentification de base préemptive avec RestTemplate et HttpClient

L'authentification de base préemptive consiste à envoyer des informations d'authentification de base HTTP (nom d'utilisateur et mot de passe) *avant* que le serveur ne réponde avec une réponse `401` les demandant. Cela peut sauver un aller-retour de demande lors de la consommation d'API REST, qui nécessitent une authentification de base.

Comme décrit dans la documentation Spring , Apache HttpClient peut être utilisé comme implémentation sous-jacente pour créer des requêtes HTTP à l'aide de `HttpComponentsClientHttpRequestFactory` . HttpClient peut être configuré pour effectuer une authentification de base préemptive .

La classe suivante étend `HttpComponentsClientHttpRequestFactory` pour fournir une authentification de base préemptive.

```
/**
 * {@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);
        }
    }
}
```

Cela peut être utilisé comme suit:

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

## Utilisation de l'authentification de base avec HttpClient de HttpComponent

L'utilisation de `HttpClient` comme `RestTemplate` sous-jacente de `RestTemplate` pour créer des requêtes HTTP permet de gérer automatiquement les requêtes d'authentification de base (une réponse http 401) lors de l'interaction avec les API. Cet exemple montre comment configurer un `RestTemplate` pour y parvenir.

```
// 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);
```

## Définition d'en-têtes sur la demande Spring RestTemplate

Les méthodes d' `exchange` de `RestTemplate` vous permettent de spécifier une `HttpEntity` qui sera

écrite dans la requête lors de l'exécution de la méthode. Vous pouvez ajouter des en-têtes (tel agent utilisateur, référent ...) à cette entité:

```
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());
     }
```

Vous pouvez également ajouter un intercepteur à votre `RestTemplate` si vous devez ajouter les mêmes en-têtes à plusieurs requêtes:

```
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());
}
```

## Résultats génériques de Spring RestTemplate

Pour laisser `RestTemplate` comprendre le générique du contenu renvoyé, nous devons définir une référence de type de résultat.

`org.springframework.core.ParameterizedTypeReference` a été introduit depuis 3.2

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

Pourrait être utile pour obtenir, par exemple, la `List<User>` d'un contrôleur.

Lire RestTemplate en ligne: https://riptutorial.com/fr/spring/topic/5896/resttemplate

# Chapitre 16: SOAP WS Consommation

## Examples

### Consommer un WS SOAP avec l'authentification de base

Créez votre propre 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);
    }
}
```

Dans votre classe de configuration WS, définissez le MessageSender que vous venez de créer:

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

Lire SOAP WS Consommation en ligne: https://riptutorial.com/fr/spring/topic/9451/soap-ws-consommation

# Chapitre 17: Source de propriété

## Examples

### Annotation

Exemple de fichier de propriétés: nexus.properties

Exemple de contenu du fichier de propriétés:

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

Exemple de configuration de fichier de contexte

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

Exemple de propriété Bean utilisant des annotations

```
@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;
}
```

Exemple de classe Constant

```
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="";
}
```

### Exemple de configuration XML à l'aide de PropertyPlaceholderConfigurer

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

Lire Source de propriété en ligne: https://riptutorial.com/fr/spring/topic/6651/source-de-propriete

# Chapitre 18: Validation du haricot JSR 303 au printemps

## Introduction

Spring a le support de validation du haricot JSR303. Nous pouvons l'utiliser pour faire la validation des beans d'entrée. Séparez la logique de validation de la logique métier à l'aide de JSR303.

## Examples

### JSR303 Validation basée sur des annotations dans des exemples de ressorts

Ajoutez toute implémentation JSR 303 à votre chemin de classe. Le plus utilisé est le validateur Hibernate de Hibernate.

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

Disons qu'il y a un api de repos pour créer un utilisateur dans le système

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

L'échantillon json d'entrée ressemblerait à celui ci-dessous

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

User.java

```
public class User {

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

    getXXX and setXXX

}
```

Nous pouvons définir les validations JSR 303 sur la classe d'utilisateur ci-dessous.

```
public class User {

    @NotEmpty
```

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

    @NotEmpty
    private String password;

    @NotEmpty
    private String password2;

}
```

Nous pouvons également avoir besoin d'un validateur d'entreprise comme le mot de passe et le mot de passe2 (confirmer le mot de passe) sont les mêmes, pour cela nous pouvons ajouter un validateur personnalisé comme ci-dessous. Ecrivez une annotation personnalisée pour annoter le champ de données.

```
@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 {};
}
```

Ecrivez une classe de validateur pour appliquer la logique de validation.

```
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());
    }
}
```

Ajout de cette validation à la classe d'utilisateurs

```
@GoodPassword
public class User {

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

    @NotEmpty
    private String password;

    @NotEmpty
    private String password2;
}
```

@Valid déclenche la validation au printemps. BindingResult est un objet injecté par Spring qui a la

liste des erreurs après validation.

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

L'annotation JSR 303 contient des attributs de message qui peuvent être utilisés pour fournir des messages personnalisés.

```
@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 - Personnalisez les messages d'erreur

Supposons que nous ayons une classe simple avec des annotations de validation

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

    @Min(18)
    private int age;

//getters/setters
}
```

Un contrôleur pour vérifier la validité 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);
    }
}
```

Et un test

```
@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"));
}
```

Le nom et l'âge ne sont pas valides. Dans le BindingResult, nous avons deux erreurs de validation. Chacun a un tableau de codes.

Codes pour vérification min.

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

Et pour le chèque NotEmpty

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

Ajoutons un fichier custom.properties pour remplacer les messages par défaut.

```
@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 nous ajoutons au fichier custom.properties la ligne

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

La nouvelle valeur est affichée pour l'erreur. Pour résoudre le message, le validateur examine les codes en commençant par le début pour trouver les messages appropriés.

Ainsi, lorsque nous définissons la clé `NotEmpty` dans le fichier .properties pour tous les cas où l'annotation `@NotEmpty` est utilisée, notre message est appliqué.

Si on définit un message

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

Toutes les annotations où nous appliquons min check à des valeurs entières utilisent le message nouvellement défini.

La même logique pourrait être appliquée si nous devons localiser les messages d'erreur de validation.

## Utilisation de @Valid pour valider les POJO imbriqués

Supposons que nous ayons un utilisateur de classe POJO que nous devons valider.

```
public class User {

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

et une méthode de contrôleur pour valider l'instance d'utilisateur

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

Étendons l'utilisateur avec une adresse POJO imbriquée que nous devons également valider.

```
public class Address {

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

Ajoutez `@Valid` annotation `@Valid` le champ d'adresse pour exécuter la validation des POJO imbriqués.

```
public class User {
```

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

    @Valid
    private Address address;
}
```

Lire Validation du haricot JSR 303 au printemps en ligne: https://riptutorial.com/fr/spring/topic/9882/validation-du-haricot-jsr-303-au-printemps

# Crédits

| S. No | Chapitres | Contributeurs |
|---|---|---|
| 1 | Commencer avec le printemps | Andrei Maieras, Community, dimitrisli, guille11, Hitesh Kumar, ipsi, Panther, Rajanikanta Pradhan |
| 2 | Base de printemps | dimitrisli, JDC, parlad neupane, Rajanikanta Pradhan |
| 3 | Comprendre le dispatcher-servlet.xml | Srinivas Gadilli |
| 4 | Configuration ApplicationContext | nicholas.hauschild |
| 5 | Créer et utiliser des haricots | CollinD, Constantine, Harshal Patil, JamesENL, Maciej Walkowiak, mszymborski, nicholas.hauschild, Panther, StanislavL, Stefan Isele - prefabware.com, Tim Tong |
| 6 | Enregistrement de haricot conditionnel au printemps | 4444, Bond - Java Bond, StanislavL |
| 7 | Étendues de haricots | Constantine, Panther, Taylor, Tim Tong, xpadro |
| 8 | Exécution des tâches et planification | guille11, Johir, Xtreme Biker |
| 9 | Initialisation paresseuse du printemps | AdamIJK, DavidR, Moshe Arad |
| 10 | Injection de dépendance (DI) et inversion de contrôle (IoC) | manish, Sergii Bishyr, walsh |
| 11 | JdbcTemplate | Setu, smichel, StanislavL |
| 12 | Langage d'expression du printemps (SpEL) | Stephen Leppik, walsh |
| 13 | Obtenir un | Arlo |

| | | |
|---|---|---|
| | SqlRowSet à partir de SimpleJdbcCall | |
| 14 | Profil de printemps | StanislavL |
| 15 | RestTemplate | bernie, eltabo, StanislavL |
| 16 | SOAP WS Consommation | guille11 |
| 17 | Source de propriété | Gautam Jose, Panther |
| 18 | Validation du haricot JSR 303 au printemps | Praneeth Ramesh, StanislavL |