eBook Gratuit

APPRENEZ spring-mvc

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

#spring-mvc

Table des matières

À propos

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

It is an unofficial and free spring-mvc 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-mvc.

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: Démarrer avec spring-mvc

Remarques

Cette section fournit une vue d'ensemble de ce qu'est spring-mvc et pourquoi un développeur peut vouloir l'utiliser.

Il devrait également mentionner tous les grands sujets dans Spring-mvc, et établir un lien avec les sujets connexes. La documentation de spring-mvc étant nouvelle, vous devrez peut-être créer des versions initiales de ces rubriques connexes.

Versions

Printemps

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

Spécification du servlet

Version	Date de sortie
3.1	2013-05-31
3.0	2009-12-31
2,5	2005-09-30
2.4	2003-11-30
2.3	2001-08-31
2.2	1999-08-31
2.1	1998-11-30
1.0	1997-07-31

Examples

Premier projet Spring-MVC

Créer un projet Web dynamique, fournir les informations suivantes comme indiqué ci-dessous

1. Nom du projet: DemoSpringMVCProject
2. Temps d'exécution cible: défini comme serveur Apache Tomcat v7.0

Cliquez sur Terminer, nous avons créé un projet Web dynamique.

Maintenant, nous devons configurer le framework Spring-MVC:

1. Créez **le fichier web.xml** sous **le dossier 'WebContent \ WEB-INF \'**

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="http://java.sun.com/xml/ns/javaee/web-
app_2_5.xsd" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" id="WebApp_ID" version="2.5">
<display-name>Demo9</display-name>

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

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

</web-app>
```

- Où classe DispatcherServlet Intercepte la demande entrante et détermine quel contrôleur

gère la demande.

- Nous allons utiliser le nom de servlet " **demo** " lors de la création de servlet.xml

2. Créez le **fichier demo-servlet.xml** sous **le dossier 'WebContent \ WEB-INF \'**

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

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

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

</beans>
```

- context: scan de composants est utilisé pour analyser tous les contrôleurs définis dans **le** package **'com'** .
- Interface ViewResolver Permet de gérer le mappage entre les vues logiques et les vues réelles. Une implémentation prédéfinie du résolveur de vue est disponible pour mapper les vues. Ex: InternalResourceViewResolver, VelocityViewResolver.
- Pour rechercher toutes les pages jsp dont nous avons défini le **préfixe** qui n'est rien d'autre que la propriété setter, sa valeur est définie comme **«/ WEB-INF / jsp /»** (chemin du dossier). **Suffixe** qui n'est rien d'autre que la propriété getter, sa valeur est définie comme **'.jsp'** (fichier de recherche avec une extension .jsp)

3. Ajouter les bibliothèques requises:

Laissez-nous ajouter Spring Framework et les bibliothèques API de journalisation communes à notre projet. Pour ce faire, cliquez avec le bouton droit de la souris sur le nom de votre projet DemoSpringMVCProject, puis suivez l'option suivante disponible dans le menu contextuel: Build Path -> Configure Build Path pour afficher la fenêtre Java Build Path comme suit:

Utilisez maintenant le bouton Ajouter des fichiers JAR externes disponibles sous l'onglet Bibliothèques pour ajouter les fichiers JAR principaux suivants dans les répertoires d'installation Spring Framework et Common Logging:

- commons-logging-1.1.1
- Spring-aop-4.1.6.RELEASE
- aspects printemps-4.1.6.RELEASE
- haricots printaniers-4.1.6.RELEASE

- spring-context-4.1.6.RELEASE
- spring-context-support-4.1.6.RELEASE
- Spring-core-4.1.6.RELEASE
- expression-printemps-4.1.6.RELEASE
- instrument à ressort-4.1.6.RELEASE
- spring-instrument-tomcat-4.1.6.RELEASE
- spring-jdbc-4.1.6.RELEASE
- spring-jms-4.1.6.RELEASE
- messagerie printemps-4.1.6.RELEASE
- spring-orm-4.1.6.RELEASE
- spring-oxm-4.1.6.RELEASE
- test de printemps-4.1.6.RELEASE
- spring-tx-4.1.6.RELEASE
- printemps-web-4.1.6.RELEASE
- spring-webmvc-4.1.6.RELEASE
- spring-webmvc-portlet-4.1.6.RELEASE
- printemps-websocket-4.1.6.RELEASE

Passons au contrôleur et aux pages jsp:

1. Créez un package **com.demo.controller** sous le dossier **src** .
2. Créer une classe **LoginController** sous paquet **com.demo.controller**

```
package com.demo.controller;

import javax.servlet.http.HttpServletRequest;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class LoginController {

@RequestMapping("/")
public String startPage(){
    return "login";
}

@RequestMapping("/signin")
```

```
public String handleRequest(HttpServletRequest request){
    String name = request.getParameter("name");
    String pass = request.getParameter("password");
    if(name.equals(pass))
    {
        return "welcome";
    }else{
        return "login";
    }

   }

}
```

3. Créez une page **login.jsp et welcome.jsp** sous **'WebContent \ WEB-INF \ jsp \'**

login.jsp

```
   <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
   <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
   <html>
   <head>
    <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
    <title>Login Page</title>
    </head>
    <body>
      <form action="signin">
          <table>
            <tr>
                <td>User Name : </td>
                <td><input type="text" name="name" id="name"/> </td>
            </tr>
            <tr>
                <td>Password: </td>
                <td><input type="text" name="password"  id="password"/> </td>
            </tr>
            <tr>
                <td colspan="2"><input type="submit" value="Login"/></td>
            </tr>
          </table>
      </form>
    </body>
    </html>
```

welcome.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
```

```
  <h1> Welcome to Spring MVC !!! </h1>
</body>
</html>
```

Ajoutez DemoSpringMVCProject dans le serveur localTomcat et exécutez-le sur le serveur.

Lire Démarrer avec spring-mvc en ligne: https://riptutorial.com/fr/spring-mvc/topic/824/demarrer-avec-spring-mvc

Chapitre 2: Gestion des exceptions

Syntaxe

- @ExceptionHandler (ExceptionToBeHandled.class)
- @ExceptionHandler ({ExceptionToBeHandled.class, AnotherExceptionToBeHandled.class})

Examples

Gestion des exceptions par contrôleur

Dans le cas où un contrôleur lève une exception, nous pouvons définir des méthodes de gestionnaire d'exceptions pour générer et renvoyer des réponses spécifiques. Il est important de noter que les gestionnaires d'exceptions définis dans le contrôleur donné ne s'appliqueront qu'aux exceptions qui se produisent dans ce contrôleur.

```
@Controller
public class MyController {
    @GetMapping("/")
    public String somePage() throws Exception {
        // some set of code that can throw exceptions
    }

    @ExceptionHandler(Exception.class)
    public String genericErrorPage() {
        return "genericErrorView";
    }

    @ExceptionHandler(ChildException.class)
    public String childErrorPage(ChildException ex) {
        return "childErrorView with msg=" + ex.getMessage();
    }
}
```

Si plusieurs gestionnaires d'exceptions sont définis, la méthode avec l'exception la plus spécifique sera choisie. Prenons le code ci-dessus comme exemple. Si une `ChildException` est levée, la méthode `childErrorPage()` sera invoquée.

Supposons qu'une `NullPointerException` soit lancée. Dans ce cas, la méthode `genericErrorPage()` sera appelée. En effet, il n'existe pas de gestionnaire d'exception spécifique défini pour `NullPointerException` , mais `NullPointerException` est une classe descendante d' `Exception` .

Cet exemple montre également comment accéder à l'exception. Dans le gestionnaire `childErrorPage` nous avons une `ChildException` transmise en tant que paramètre. Il est alors disponible pour être utilisé dans le corps du gestionnaire, comme indiqué. De même, vous pouvez définir ce gestionnaire comme ceci:

```
    @ExceptionHandler(ChildException.class)
    public String childErrorPage(HttpServletRequest req, ChildException ex) {
```

```
        // Both request and exception objects are now available
        return "childErrorView with msg=" + ex.getMessage();
    }
```

Cela vous permet d'accéder à la requête qui a généré l'exception ainsi qu'à l'exception qui a été déclenchée.

Lire Gestion des exceptions en ligne: https://riptutorial.com/fr/spring-mvc/topic/6202/gestion-des-exceptions

Chapitre 3: Gestion des exceptions globales

Remarques

1. N'oubliez pas de créer des exceptions personnalisées si vous devez
2. Le résolveur et le gestionnaire doivent tous deux être découverts par Spring
3. Si vous êtes sur Spring 3.2 ou supérieur, vous pouvez utiliser `@ContrllerAdvice`

La source

Examples

Résolveur d'exception global

```
@Component
public class RestExceptionResolver extends ExceptionHandlerExceptionResolver {

    @Autowired
    //If you have multiple handlers make this a list of handlers
    private RestExceptionHandler restExceptionHandler;
    /**
     * This resolver needs to be injected because it is the easiest (maybe only) way of
getting the configured MessageConverters
     */
    @Resource
    private ExceptionHandlerExceptionResolver defaultResolver;

    @PostConstruct
    public void afterPropertiesSet() {
        setMessageConverters(defaultResolver.getMessageConverters());
        setOrder(2); // The annotation @Order(2) does not work for this type of component
        super.afterPropertiesSet();
    }

    @Override
    protected ServletInvocableHandlerMethod getExceptionHandlerMethod(HandlerMethod
handlerMethod, Exception exception) {
        ExceptionHandlerMethodResolver methodResolver = new
ExceptionHandlerMethodResolver(restExceptionHandler.getClass());
        Method method = methodResolver.resolveMethod(exception);
        if (method != null) {
            return new ServletInvocableHandlerMethod(restExceptionHandler, method);
        }
        return null;
    }

    public void setRestExceptionHandler(RestExceptionHandler restExceptionHandler) {
        this.restExceptionHandler = restExceptionHandler;
    }

    public void setDefaultResolver(ExceptionHandlerExceptionResolver defaultResolver) {
        this.defaultResolver = defaultResolver;
    }
}
```

Alors un exemple de gestionnaire ressemblera à ceci

```
@Component
public class RestExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    @ResponseStatus(HttpStatus.NOT_FOUND)
    @ResponseBody
    public Map<String, Object> handleException(ResourceNotFoundException e,
HttpServletResponse response) {
        Map<String, Object> error = new HashMap<>();
        error.put("error", e.getMessage());
        error.put("resource", e.getResource());
        return error;
    }
 }
```

Bien sûr, vous n'oublierez pas d'enregistrer vos beens

Lire Gestion des exceptions globales en ligne: https://riptutorial.com/fr/spring-mvc/topic/7648/gestion-des-exceptions-globales

Chapitre 4: Spring-MVC avec des annotations

Introduction

Dans cette rubrique, vous découvrirez les annotations principalement liées à Spring MVC. Certaines des annotations associées sont les suivantes: `@Controller` , `@RequestMapping` , `@RequestParam` , `@RequestBody` , `@ResponseBody` , `@RestController` , `@ModelAttribute` , `@ControllerAdvice` , `@ExceptionHandler` , `@ResponseStatus` .

Bien sûr, il y a plus d'annotations qui sont extrêmement importantes mais qui n'appartiennent pas directement à Spring MVC. Tels que: `@Required` , `@Autowired` , `@Resource` et bien d'autres.

Paramètres

Annotation	Explication
`@Controller`	Avec l'annotation `@Controller` vous marquez une classe Java en tant que classe `@Controller` des gestionnaires HTTP, en d'autres termes, des points d'accès HTTP à votre application.
`@RequestMapping`	L'annotation `@RequestMapping` est celle que vous utiliserez pour marquer les gestionnaires HTTP (points d'accès HTTP à votre application) dans votre classe `@Controller`
`@RequestParam`	Utilisez l'annotation `@RequestParam` pour lier les paramètres de requête à un paramètre de méthode de votre contrôleur.

Examples

dispatcher-servlet.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:mvc="http://www.springframework.org/schema/mvc"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc-4.3.xsd
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.3.xsd">


    <mvc:annotation-driven/>
    <context:component-scan base-package="your.base.package.to.scan" />
</beans>
```

Avec ces deux lignes de configuration, vous pourrez utiliser les annotations MVC.

@Controller & @RequestMapping

```
@Controller
@RequestMapping("/appointments")
public class AppointmentsController {

//your handlers here, for example:

@RequestMapping(path = "/new", method = RequestMethod.GET)
public AppointmentForm getNewForm() {
    return new AppointmentForm();
}

@RequestMapping(method = RequestMethod.POST)
public String add(@Valid AppointmentForm appointment, BindingResult result) {
    if (result.hasErrors()) {
        return "appointments/new";
    }
    appointmentBook.addAppointment(appointment);
    return "redirect:/appointments";
}
```

}

Avec l'annotation `@Controller` vous marquez une classe Java en tant que classe `@Controller` plusieurs gestionnaires HTTP, en d'autres termes, des points d'accès HTTP à votre application.

L'annotation `@RequestMapping` est celle que vous utiliserez pour marquer les gestionnaires HTTP (points d'accès HTTP à votre application) dans votre classe @Controller.

@RequestParam

```
@Controller
public class EditPetForm {

    @RequestMapping("/pets")
    public String setupForm(@RequestParam("petId") int petId, ModelMap model) {
        Pet pet = this.clinic.loadPet(petId);
        model.addAttribute("pet", pet);
        return "petForm";
    }
}
```

Il est important de mentionner, mais assez évident, que `@RequestParam` est conçu pour fonctionner avec la méthode HTTP GET uniquement parce que seul GET vous permet d'envoyer une chaîne de requête avec des paramètres et que `@RequestParam` vous lie les paramètres de la chaîne de requête à votre gestionnaire de contrôleur paramètres.

Lire Spring-MVC avec des annotations en ligne: https://riptutorial.com/fr/spring-mvc/topic/10662/spring-mvc-avec-des-annotations

Chapitre 5: Téléchargement de fichiers

Syntaxe

- `@RequestPart(String, String, boolean)`

Paramètres

Paramètre	Détails
`@RequestPart`	Cette annotation spécifie qu'un paramètre doit être mappé à une partie de requête donnée. Le nom de la pièce doit correspondre au nom du paramètre de la méthode, *sauf si* vous choisissez de le fournir en tant qu'argument à `@RequestPart` . Si le nom de la pièce n'est pas exprimable en tant que nom Java (par exemple `123`), vous pouvez utiliser l'attribut `value` de `@RequestPart` pour spécifier le nom réel. par exemple `@RequestPart("123") String _123` .

Remarques

Si vous utilisez une ancienne version de Java (version 1.7) ou si vous compilez *sans* informations de débogage, Java remplacera le nom du paramètre par `arg0` , `arg1` , etc., ce qui empêchera Spring de les associer à les noms des pièces. Si tel est le cas, vous devrez alors définir le nom de la pièce dans l'annotation `@RequestPart` , comme `@RequestPart` dans la `@RequestPart` Paramètres.

Examples

Téléchargement d'un fichier unique

Pour recevoir un fichier téléchargé via un message HTTP, vous devez procéder comme suit:

```
@RequestMapping(
    value = "...",
    method = RequestMethod.POST,
    consumes = MediaType.MULTIPART_FORM_DATA_VALUE
)
public Object uploadFile(
    @RequestPart MultipartFile file
) {
    String fileName = file.getOriginalFilename();
    InputStream inputStream = file.getInputStream();
    String contentType = file.getContentType();
    .
    .
    .
}
```

Notez que le nom du paramètre `@RequestPart` doit correspondre au nom de la pièce dans la demande.

En HTML:

```
 <form action="/..." enctype="multipart/form-data" method="post">
        <input type="file" name="file">
    </form>
```

En HTML (Spring TagLibs):

```
<form action="/..." enctype="multipart/form-data" method="post">
    <form:input type="file" path="file">
</form>
```

En tant que requête HTTP brute:

```
POST /... HTTP/1.1
Host: ...
Content-Type: multipart/form-data; boundary=----------287032381131322

------------287032381131322
Content-Disposition: form-data; name="file"; filename="r.gif"
Content-Type: image/gif

GIF87a.............,..........D..;
------------287032381131322--
```

Cette demande signifierait ce qui suit:

```
fileName == "r.gif"
contentType == "image/gif"
```

Au printemps MVC

Besoin d'ajouter le bean mentionné pour accéder à la fonctionnalité multi-parties

```
    <!-- max size of file in memory (in bytes) -->
    <property name="maxInMemorySize" value="1048576" /> <!-- 1MB -->

</bean>
```

Téléchargement de plusieurs fichiers

Pour recevoir plusieurs fichiers téléchargés via une seule publication HTTP, vous devez procéder comme suit:

```
@RequestMapping(
    value = "...",
    method = RequestMethod.POST,
    consumes = MediaType.MULTIPART_FORM_DATA_VALUE
)
```

```
public Object uploadFile(
    @RequestPart MultipartFile[] files
) {
    for (file : files) {
        String fileName = file.getOriginalFilename();
        InputStream inputStream = file.getInputStream();
        String contentType = file.getContentType();
        .
        .
        .
    }
}
```

Notez que le nom du paramètre `@RequestPart` doit correspondre au nom de la pièce dans la demande.

En HTML:

```
<form action="/..." enctype="multipart/form-data" method="post">
    <input type="file" name="files">
    <input type="file" name="files">
</form>
```

En tant que requête HTTP brute:

```
POST /... HTTP/1.1
Host: ...
Content-Type: multipart/form-data; boundary=----------287032381131322

------------287032381131322
Content-Disposition: form-data; name="files"; filename="r.gif"
Content-Type: image/gif

GIF87a.............,..........D..;
------------287032381131322
Content-Disposition: form-data; name="files"; filename="banana.jpeg"
Content-Type: image/jpeg

GIF87a.............,..........D..;
------------287032381131322--
```

Cette demande signifierait ce qui suit:

```
files[0].getOriginalFilename() == "r.gif"
files[0].getContentType() == "image/gif"
files[1].getOriginalFilename() == "r.jpeg"
files[1].getContentType() == "image/jpeg"
```

Téléchargement de plusieurs pièces avec des noms différents

Il est possible de télécharger plusieurs pièces, chacune avec un nom différent. Pour chaque nom de pièce, vous aurez besoin d'un paramètre annoté avec `@RequestPart` , dont le nom correspond au nom de la pièce.

Pour recevoir un fichier téléchargé via un message HTTP, vous devez procéder comme suit:

```
@RequestMapping(
    value = "...",
    method = RequestMethod.POST,
    consumes = MediaType.MULTIPART_FORM_DATA_VALUE
)
public Object uploadFile(
    @RequestPart MultipartFile profilePicture,
    @RequestPart MultipartFile companyLogo,
) {
    .
    .
    .
}
```

En HTML:

```
<form action="/..." enctype="multipart/form-data" method="post">
    <input type="file" name="profilePicture">
    <input type="file" name="companyLogo">
</form>
```

En tant que requête HTTP brute:

```
POST /... HTTP/1.1
Host: ...
Content-Type: multipart/form-data; boundary=----------287032381131322

------------287032381131322
Content-Disposition: form-data; name="profilePicture"; filename="r.gif"
Content-Type: image/gif

GIF87a.............,..........D..;
------------287032381131322
Content-Disposition: form-data; name="companyLogo"; filename="banana.jpeg"
Content-Type: image/jpeg

GIF87a.............,..........D..;
------------287032381131322--
```

Marshaling une pièce dans un objet

Si vous souhaitez convertir le contenu d'une pièce en objet de domaine (par exemple, un `User` un `Account` ou une `Address`), le processus est très simple:

Il est possible de télécharger plusieurs pièces, chacune avec un nom différent. Pour chaque nom de pièce, vous aurez besoin d'un paramètre annoté avec `@RequestPart` , dont le nom correspond au nom de la pièce.

Pour recevoir un fichier téléchargé via un message HTTP, vous devez procéder comme suit:

```
@RequestMapping(
    value = "...",
```

```
    method = RequestMethod.POST,
    consumes = MediaType.MULTIPART_FORM_DATA_VALUE
)
public Object uploadFile(
    @RequestPart Address address,
) {
    .
    .
    .
}
```

En tant que requête HTTP brute:

```
POST /... HTTP/1.1
Host: ...
Content-Type: multipart/form-data; boundary=----------287032381131322

------------287032381131322
Content-Disposition: form-data; name="address"; filename="address.json"
Content-Type: application/json

{"houseNumber": "10/A", "streetName": "Dumbldore Road", "town": "Hogsmede"}
------------287032381131322--
```

Les choses les plus importantes sont:

- Le nom de la pièce doit correspondre au nom de la variable.
- Le `Content-Type` de la pièce doit être celui que Spring serait capable de gérer si vous l'aviez envoyé en tant que demande régulière. Autrement dit, si vous pouviez exécuter un `POST` sur un noeud final avec un `Content-Type` de `foo/bar` et que Spring est capable de le transformer en objet, il sera également capable de rassembler une partie dans un objet.
- Vous *devez* pouvoir définir le `Content-Type` de `Content-Type` de la pièce. Si vous ne pouvez pas, cette approche ne fonctionnera pas - Spring *ne* tentera *pas* de deviner le `Content-Type` de `Content-Type` de la pièce.

Lire Téléchargement de fichiers en ligne: https://riptutorial.com/fr/spring-mvc/topic/3050/telechargement-de-fichiers

Chapitre 6: Validation Spring MVC

Remarques

Dans Spring MVC, il est possible de valider les champs de formulaire à l'aide de l'API Bean Validation (JSR 303 pour Bean Vaidation 1.0 et JSR 349 pour la validation Bean 1.1), utilisés pour définir les contraintes de validation de l'objet JavaBean.

Hibernate Validator est l'implémentation de référence de l'API Bean Validation. Hibernate Validator offre une valeur supplémentaire par rapport aux fonctionnalités requises par la validation Bean. Par exemple, une API de configuration de contraintes par programme ainsi qu'un processeur d'annotations qui se connecte au processus de génération et génère des erreurs de compilation lorsque des annotations de contraintes sont utilisées de manière incorrecte.

Examples

Spring MVC Validation de formulaire avec API de validation de bean

Cet exemple montre comment valider les formulaires dans Spring MVC à l'aide de l' **API Bean Validation à l'** aide d'annotations Java, sans aucun `xml` . L'utilisateur sera proposé de saisir ses données d'enregistrement et le validateur vérifiera sa validité.

Ajouter des dépendances

Ajoutez tout d'abord les dépendances suivantes dans votre projet:

- API de validation de bean et
- Moteur de validation Hibernate

```
dependencies {
    compile group: 'javax.validation', name: 'validation-api', version: '1.1.0.Final'
    compile group: 'org.hibernate', name: 'hibernate-validator', version: '5.2.4.Final'
}
```

Créer une classe de modèle

Créez la classe de modèle `User` comme ci-dessous:

```
import org.hibernate.validator.constraints.Email;
import org.springframework.format.annotation.DateTimeFormat;

import javax.validation.constraints.Max;
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Past;
import javax.validation.constraints.Size;
```

```
import java.util.Date;

public class User {

    @NotNull(message = "Please input your email.")
    @Email(message = "Email format is wrong.")
    private String email;

    @NotNull(message = "{user.password.notNull}")
    @Size(min = 8, max = 16, message = "{user.password.size}")
    private String password;

    @NotNull(message = "{user.age.notNull}")
    @Min(18)
    @Max(100)
    private Integer age;

    @NotNull(message = "{user.birthday.notNull}")
    @DateTimeFormat(pattern = "dd.MM.yyyy")
    @Past(message = "{user.birthday.past}")
    private Date birthday;

    // getters, setters
}
```

Voici certaines des annotations JSR 303: `@NotNull` , `@Size` , `@Min` , `@Max` et `@Past` , ainsi que des annotations supplémentaires fournies par l'implémentation du validateur d'hibernation: `@Email` , `@DateTimeFormat` .

Notez que les messages d'erreur pour `email` champ `email` sont spécifiés dans ses annotations. Alors que les messages d'erreur pour les champs `password` , `age` et `birthday` sont spécifiés dans un fichier `messages.properties` afin de démontrer l' *externalisation des messages d'erreur de validation* . Ces fichiers doivent être placés dans le dossier de `resources` :

```
user.password.notNull = Password field cannot be empty.
user.password.size = Password must be between {min} and {max} characters in length.
user.age.notNull = Please enter your age.
user.birthday.notNull = Please enter your birthday.
user.birthday.past = That's impossible.

typeMismatch=Please use dd.MM.yyyy format
```

Pour cette possibilité, `messageSource()` avec `bean.setBasename("classpath:messages");` Les beans code et `validator()` doivent également être configurés et annotés:

```
@Configuration
@PropertySource("application.properties")
public class AppConfig extends WebMvcConfigurerAdapter {

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

```
    }

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

    @Override
    public Validator getValidator() {
        return validator();
    }
}
```

De même, la classe de configuration doit être annotée avec `@PropertySource("application.properties")` et le chemin d'accès aux pages `jsp` doit être ajouté à ce fichier comme suit:

```
spring.mvc.view.prefix=/WEB-INF/jsp/
spring.mvc.view.suffix=.jsp
```

Créer une classe FormController

Or , dans la classe de contrôleur, annoter l'objet modèle qui soutient le formulaire par la `@Valid` annotation de `javax.validation` paquet.

Spring MVC valide l'objet de modèle annoté par l'annotation `@Valid` après la liaison de ses propriétés avec les entrées du formulaire JSP qui utilise les balises de formulaire Spring. Toute violation de contrainte sera exposée en tant `BindingResult` dans l'objet `BindingResult` . Nous pouvons donc vérifier la violation dans la méthode du contrôleur.

```
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

import javax.validation.Valid;
import java.util.HashMap;
import java.util.Map;

@Controller
public class FormController {

    private Map<String, User> users = null;

    public FormController() {
        users = new HashMap<String, User>();
    }

    @RequestMapping(value = "/", method = RequestMethod.GET)
    public String viewRegister(Map<String, Object> model) {
        User user = new User();
        model.put("user", user);
```

```
        return "register";
    }

    @RequestMapping(value = "/register", method = RequestMethod.POST)
    public String doRegister(@Valid User user, BindingResult result, Model model) {
        if (result.hasErrors()) {
            return "register";
        }
        model.addAttribute("user", user);
        users.put(user.getEmail(), user);
        return "registerSuccess";
    }
}
```

Créer un formulaire de saisie JSP

Ajoutez le fichier `register.jsp` avec le contenu suivant:

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>User Form Page</title>
<style>
.error {
    color: #ff0000;
    font-weight: bold;
}
</style>
</head>
<body>
    <form:form method="POST" commandName="user" action="register">
        <table>
            <tr>
                <td>Email:</td>
                <td><form:input path="email" placeholder="Email"/></td>
                <td><form:errors path="email" cssClass="error" /></td>
            </tr>
            <tr>
                <td>Password:</td>
                <td><form:password path="password" placeholder="Password"/></td>
                <td><form:errors path="password" cssClass="error" /></td>
            </tr>
            <tr>
                <td>Age:</td>
                <td><form:input path="age" placeholder="Age"/></td>
                <td><form:errors path="age" cssClass="error" /></td>
            </tr>
            <tr>
                <td>Birthday:</td>
                <td><form:input path="birthday" placeholder="dd.MM.yyyy"/></td>
                <td><form:errors path="birthday" cssClass="error" /></td>
            </tr>
            <tr>
                <td colspan="3"><input type="submit" value="Register"></td>
```

```
            </tr>
        </table>

    </form:form>

</body>
</html>
```

En règle générale, nous renverrons le formulaire de saisie à l'utilisateur lorsque des erreurs de validation se sont produites. Et dans le formulaire JSP, nous pouvons afficher des messages d'erreur de validation en utilisant la balise error du formulaire Spring comme `<form:errors path="email"/>` .

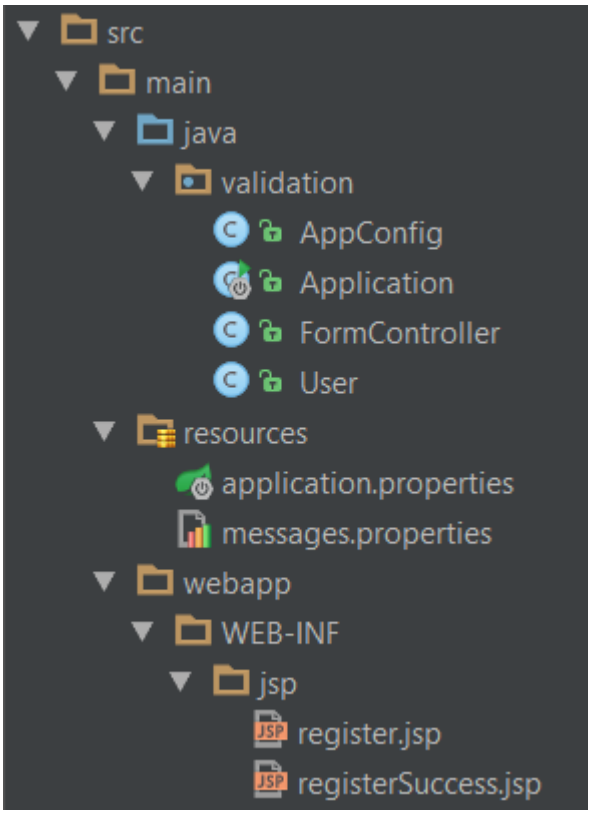
Créer une page de réussite JSP

La page `registerSuccess.jsp` sera affichée si l'utilisateur saisit toutes les données valides. Voici le code:

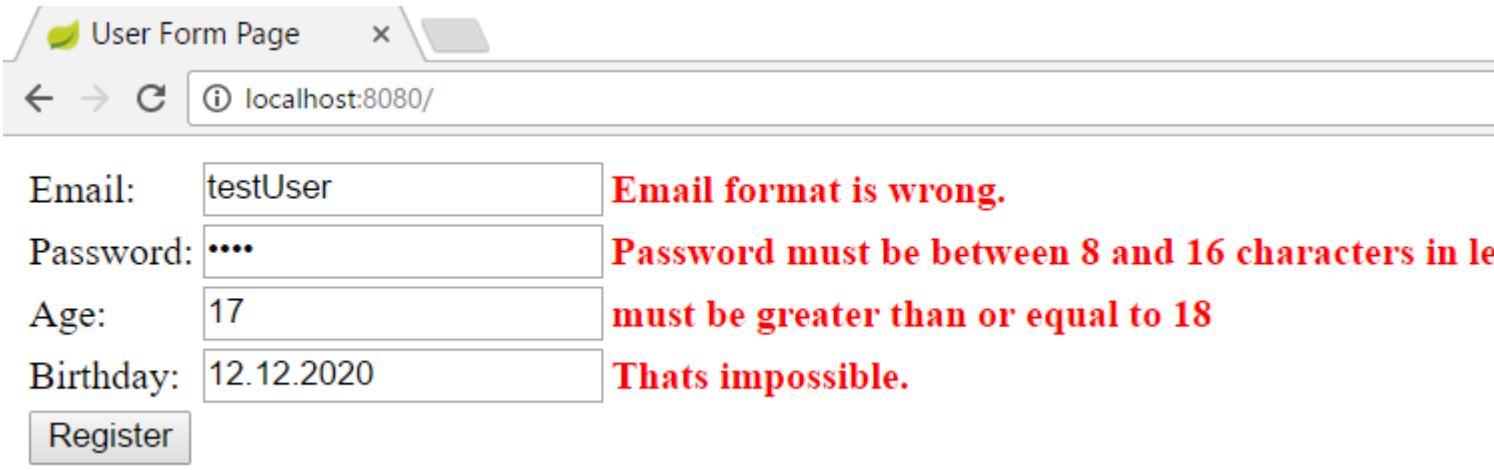
```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>

<%@ page session="false" %>
<html>
<head>
    <title>Success</title>
</head>
<body>
<h3>User Registered Successfully.</h3>

<strong>User Email: ${user.email}</strong><br>
<strong>User Age: ${user.age}</strong><br>
<strong>User Birthday: <fmt:formatDate value="${user.birthday}" type="date"
pattern="dd.MM.yyyy"/></strong><br>

</body>
</html>
```

Application de test

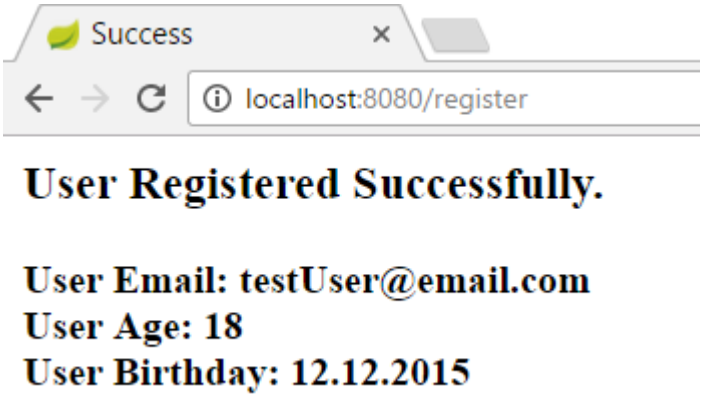
Après tout la structure du projet devrait ressembler à ceci:



Démarrez l'application, accédez à `http://localhost:8080/` et essayez d'entrer des données non valides:



Lorsque des données valides sont entrées, l'utilisateur redirige vers la page de succès:



Lire Validation Spring MVC en ligne: https://riptutorial.com/fr/spring-mvc/topic/7127/validation-spring-mvc

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec spring-mvc	Community, Flash, ipsi, Slava Semushin
2	Gestion des exceptions	Nathaniel Ford, Tim Tong
3	Gestion des exceptions globales	Amanuel Nega
4	Spring-MVC avec des annotations	Moshe Arad
5	Téléchargement de fichiers	akhilsk, ipsi
6	Validation Spring MVC	DimaSan