JAVA/J2EE au plus haut niveau


# Introduction au module SPRING MVC

par Samir Miloudi

Ecoalis Consulting – samir.miloudi@gmail.com

# Programme

**Partie 1 :** Introduction à Spring MVC

**Partie 2 :** Contrôleurs

**Partie 3 :** Mapping des URL

**Partie 4 :** Validation de formulaire

**Partie 5 :** Mécanisme de sélection de la vue

**Partie 6 :** Technologies de présentation

**Partie 7 :** Gestion des exceptions

# Partie 1

# Introduction à Spring MVC

- Rappels sur MVC
- Présentation de Spring MVC
- Configuration

## Rappels sur MVC

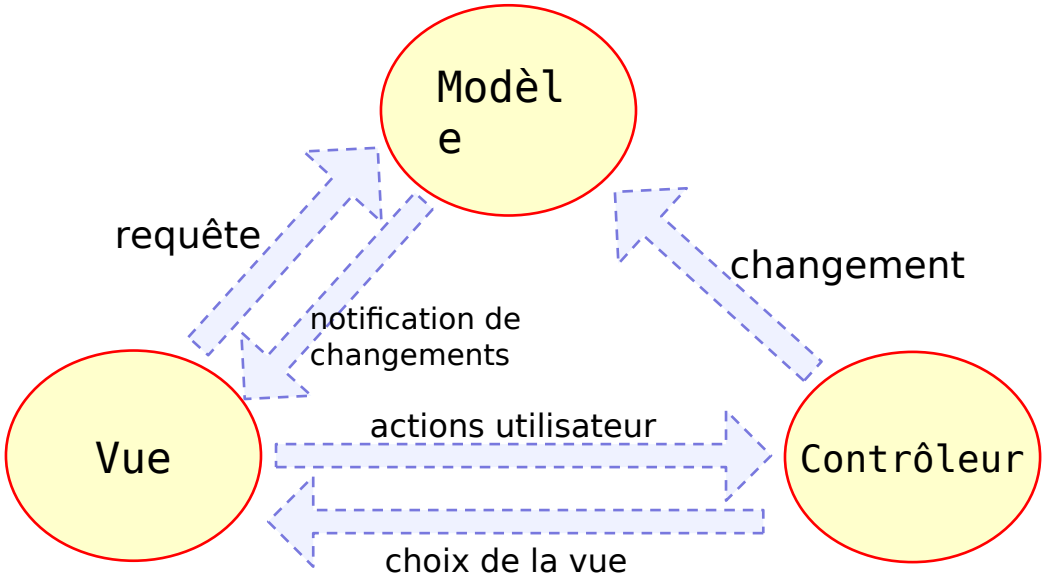
- Définition
- Schéma MVC2
- Composants mis en œuvre

## Définition

Le design pattern MVC (Model View Controller) propose une structuration du tiers présentation des application Web ou client lourds qui dissocie les traitements de la présentation proprement dite.

Le « type 2 » de ce patron de conception instaure un point d’entrée unique (contrôleur) ayant pour mission d’aiguiller les requêtes vers la bonne entité de traitement.

# Schéma MVC 2


Modèle
requête
changement
notification de changements
Vue
actions utilisateur
Contrôleur
choix de la vue

Les composants mis en œuvre dans ce pattern sont les suivants :

- **Modèle** : représente le comportement de l'application (traitements des données, interactions avec la base de données, ...) Il décrit ou contient les données manipulées par l'application. Il assure la gestion de ces données et garantit leur intégrité.

- **Vue** : correspond à l'interface avec laquelle l'utilisateur interagit et permet donc la présentation des données du modèle.

- **Contrôleur** : Gère les interactions avec le client et déclenche les traitements appropriés en interagissant avec le modèle.

# Présentation de Spring MVC

- Introduction
- Implémentation MVC de Spring
- Avantages

# Introduction (1/2)

Spring framework fournit un support pour l'intégration avec d’autres frameworks MVC, tels JSF, WebWork ou Tapestry, mais également son propre framework, **Spring MVC**.

# Introduction (2/2)

Spring MVC est un framework MVC « `request-driven` », organisé autour d'un contrôleur frontal, ou façade :

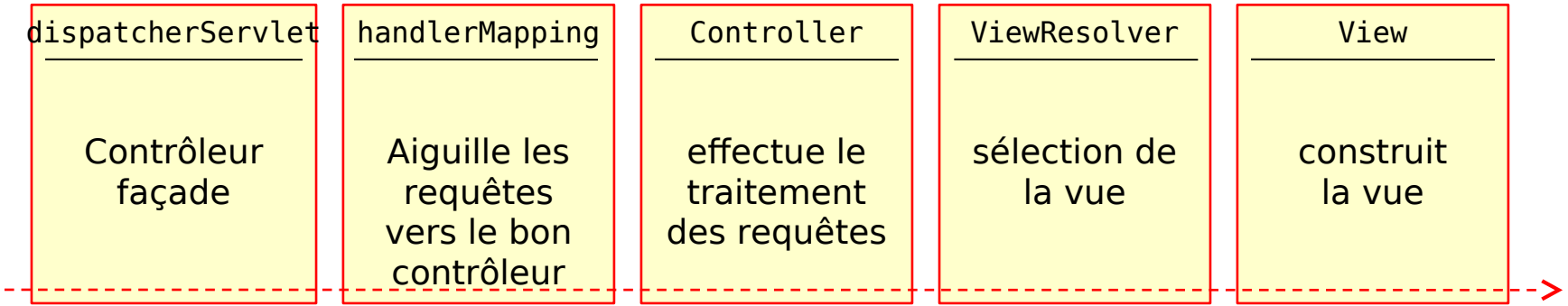
le « `DispatcherServlet` ».

Cette *servlet* délègue le traitement des requêtes aux contrôleurs selon la stratégie d'aiguillage définie (`HandlerMapping`).

Spring MVC propose également un mécanisme pour la résolution des vues, et l'intégration de technologies tierces.

# Implémentation MVC de Spring

Le traitement complet d'une requête passe par toutes ces entités :




le contrôleur façade récupère l'ensemble des requêtes utilisateur et les dispatche aux contrôleurs, qui sont des singletons issus d'une fabrique Spring.

# Avantages (1/2)

Les avantages principaux de Spring MVC sont les suivants :

- séparation claire des rôles (contrôleur, validateur, formulaires, résolveur de vues, ...)
- utilisation du conteneur léger de Spring afin de configurer les différentes entités utiles au MVC (et donc bénéficier de la résolution des dépendances)

## Avantages (2/2)

- facilitation du découplage des différentes entités mises en œuvre grâce à la programmation par interface
- utilisation d'une hiérarchie de contextes applicatifs afin de séparer logiquement des différents composants de l'application. Par exemple, les composants des services métier et des couches inférieures n'ont pas accès à ceux du MVC.

## Configuration

- Chargement du contexte de l'application Web
- Initialisation du contrôleur façade : `DispatcherServlet`

# Chargement du contexte de l'application Web

web.xml

```
<web-app>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/applicationContext.xml</param-value>
    <param-value>/WEB-INF/applicationContext-services.xml</param-value>
    <param-value>/WEB-INF/applicationContext-XXX.xml</param-value>
  </context-param>

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

</web-app>
```

Le ou les fichiers de configuration sont précisés

La classe d'écouteur ContextLoaderListener de Spring est déclarée

# Initialisation du contrôleur façade : DispatcherServlet

web.xml

```
<web-app>

  <servlet>
    <servlet-name>spring-mvc-webapp</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>2</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>spring-mvc-webapp</servlet-name>
    <url-pattern>*.form</url-pattern>
  </servlet-mapping>

</web-app>
```

1

2

1) déclare la *servlet* frontale (très similaire à Struts)
2) Mapping de *servlet* : Toutes les requêtes se terminant par `.form` seront servis par la *servlet* frontale

# Partie 2

# Contrôleurs

- Déclaration
- Implémentation

# Déclaration des contrôleurs

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:p="http://www.springframework.org/schema/p"
      xmlns:context="http://www.springframework.org/schema/context"
      xsi:schemaLocation="http://www.springframework.org/schema/bea
        http://www.springframework.org/schema/beans/spring-beans-2.
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-2.5.xsd">

  <context:component-scan base-package="com.ecoalis.springmvc.we" />  (1)

  <bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAd  (2) er"/>

  <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver"
p:prefix="/WEB-INF/jsp/" p:suffix=".jsp"/>  (3)

</beans>
```

1) Les contrôleurs seront automatiquement détectés dans le package spécifié grâce à l'annotation `@Controller`.
2) Indique que le mapping sera indiqué dans les contrôleurs par des annotations
3) Précise la manière dont les vues retournées seront sélectionnées

# Implémentation des contrôleurs (1/2)

Spring n'impose pas que les contrôleurs héritent d'une classe particulière.

Il suffit de les annoter avec `@Controller`, ou d'implémenter l'interface `org.springframework.web.servlet.mvc.Controller`.

# Implémentation des contrôleurs (2/2)

Il est également possible d'étendre l'une des nombreuses implémentations fournies, comme par exemple :

- `MultiActionController`, qui agrège plusieurs traitements dans un même contrôleur
- `SimpleFormController`, qui injecte dans l'objet métier les paramètres de la requête

# Partie 3

# Mapping des URL

- Annotation @RequestMapping
- Exemple de mapping
- URI Template
- Annotation @PathVariable
- Signature des méthodes de traitement

- Annotation @RequestParam
- Annotation @ModelAttribute
- Annotation @ModelAttribute
- Annotation @RequestHeader
- Annotation @InitBinder

# Annotation @RequestMapping

L'annotation @RequestMapping, au niveau de la classe ou de ses méthodes, précisera quels sont les traitements de requêtes réalisés par le contrôleur.

| propriété | type | description |
|---|---|---|
| method | String[] | Spécifie la ou les méthodes HTTP supportées par le mapping. |
| params | String[] | Permet de réaliser un mapping plus fin en se basant sur les paramètres de la requête. La présence ou la non-présence (avec l'opérateur !) d'un paramètre peut être utilisée. |
| value | String[] | Correspond à la valeur de l'annotation. Cette propriété permet de définir la ou les valeurs définissant le mapping de l'élément. |

# Exemple de mapping

```
@Controller
@RequestMapping("/dummy")
public class DummyController {

    @RequestMapping(
        value={"/*/foo", "/bar"),
        method=RequestMethod.GET
        params={"admin=true", "refresh", "!authentication"}
    public void dummy() {
        [...]
    }
}
```

1) Toutes les URL en « /dummy/*/foo » et « /dummy/bar » seront traitées par cette méthode du contrôleur
2) ... si les valeurs des paramètres de la requête correspondent : admin doit être à vrai, refresh présent et authentication absent
3) Remarquer le mapping « /dummy » défini globalement pour le contrôleur et donc toutes ses méthodes

## URI Template – définition

Les modèles d'URI (Uniform Resource Identifier) permettent, lors du mapping, d'extraire des variables automatiquement.

## URI Template – définition

Ainsi, si une URL d'accès est :

```
http://www.ecoalis.com/users/john/home
```

et qu'un *template* est défini par :

```
http://www.
ecoalis.com/users/{userName}/{location}
```

Alors la variable `userName` sera remplacée par `john` et la variable `location` par `home`.

## Annotation @PathVariable

Cette annotation permet d'extraire les variables des *URI Templates* lors du mapping et les convertit dans le bon type.

```
@Controller
@RequestMapping("/users/{userName}")
public class UserController {

     @RequestMapping(value="/{location}", method=RequestMethod.GET)
     public String showUser(@PathVariable String userName, @PathVariable String location, Model model) {

       User user = userService.getUser(userName);
       Location location = locationService.getLocation(location);

       [...]

       return "showUser";
     }
```

Les variables `userName` et `location` seront positionnées automatiquement par Spring avec les valeurs contenues dans l'URI et pourront ainsi être utilisées dans le corps de la méthode du contrôleur.

## Signature des méthodes de traitement

Spring permet une grande souplesse dans l'écriture des méthodes de traitement des URL annotées avec `@RequestMapping`, à la fois pour les paramètres d'entrée mais aussi pour les types de retours.

# Paramètres d'entrée supportés (1/2)

| type | description |
|---|---|
| (Http)ServletRequest/ Response | Requêtes et Réponses de l'API Servlet |
| HttpSession | Session de l'initiateur de la requête par l'intermédiaire de l'API Servlet |
| WebRequest ou NativeWebRequest | Accès d'une manière générique aux paramètres de la requête sans utiliser l'API Servlet |
| Locale | Couple pays et langue associé à la requête |
| InputStream ou Reader | Flux d'entrée associé à la requête afin d'avoir accès au contenu de la requête |
| OutputStream ou Writer | Flux de sortie associé à la réponse de la requête afin de générer le contenu de la réponse |

# Paramètres d'entrée supportés (2/2)

| type | description |
|---|---|
| `Map, Model ou ModelMap` | Modèle utilisé pour les données présentées dans la vue. Permet l'accès aux données contenues dans le modèle et son enrichissement |
| `@RequestParam` | Paramètre de la requête dont l'identifiant est celui spécifié dans l'annotation |
| `@PathVariable` | Paramètre permettant l'accès aux variables de l'URI template |
| `@RequestHeader` | Paramètre permettant l'accès aux valeurs du header de la requête |
| `@RequestBody` | Paramètre permettant l'accès aux valeurs du corps de la requête |

# Types de retour supportés (1/2)

| type | description |
|---|---|
| `void` | Spring détermine automatiquement le nom de la vue à retourner en se basant sur une implémentation de l'interface `RequestToViewNameTranslator`. |
| `Map ou Model` | Objet contenant les données du modèle à utiliser dans la vue. L'identifiant de la vue est implicitement déduit par Spring MVC comme dans le cas de `void` |
| `ModelAndView` | Objet regroupant l'identifiant de la vue à utiliser suite aux traitements du contrôleur et le contenu du modèle pour cette dernière |

# Types de retour supportés (2/2)

| type | description |
|---|---|
| `String` | Nom logique (identifiant) de la vue à utiliser suite aux traitements du contrôleur |
| `View` | Vue à utiliser suite aux traitements du contrôleur |
| `@ModelAttribute` | Objet à ajouter aux données du modèle après l'exécution de la méthode et avant celle de la vue. L'identifiant utilisé dans l'ajout correspond à celui de l'annotation. |

# Annotation @RequestParam

Cette annotation permet de charger le paramètre de la méthode avec le paramètre correspondant de la requête.
La conversion dans le bon type de données est implicite.
Par défaut les paramètres sont requis (`required=true`).

```
@RequestMapping(method=RequestMethod.GET)
public String showUser(@RequestParam int userId, @RequestParam(required=false) String userName)
{

    [...]
}
```

les variables `userId` et `userName` sont initialisées à partir des valeurs présentes dans la requête et disponibles dans le corps de la méthode du contrôleur.

## Annotation @ModelAttribute (1/2)

Cette annotation présente deux utilisations :

1)située au niveau des paramètres d'une méthode, elle permet de récupérer une référence sur un objet contenu dans le formulaire

2)située au niveau de la méthode, elle permet de peupler les données d'un formulaire

# Annotation @ModelAttribute

```
@ModelAttribute("users")  1
public Collection<User> populateUsers() {
    return userService.getUsers();
}

@RequestMapping(method = RequestMethod.POST)
public String processSubmit(
        @ModelAttribute("user") User user,  2
        BindingResult result, SessionStatus status) {

    [...]
  userService.addUser(user);
  [...]
}
```

1) la collection d'utilisateur du formulaire sera initialisée par cette méthode
2) l'utilisateur présent dans le formulaire sera disponible dans le corps de la méthode pour traitement

# Annotation @RequestHeader

Cette annotation permet de récupérer toutes les valeurs du header de la requête comme dans l'exemple suivant :

```
@RequestMapping("/displayHeaderInfo.do")
public void displayHeaderInfo(@RequestHeader("Accept-Encoding") String encoding,
                              @RequestHeader("Keep-Alive") long keepAlive) {
   [...]
}
```

# Annotation `@InitBinder` (1/2)

La conversion effectuée automatiquement par Spring pour les paramètres des requêtes n'est pas toujours suffisante ou satisfaisante.

L'annotation `@InitBinder` permet d'enrichir les conversions des types de données.

# Annotation @InitBinder (2/2)

Dans cet exemple, un convertisseur personnalisé pour les objets de type « Date » est enregistré :

```
@InitBinder
public void initBinder(WebDataBinder binder) {

    SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
    dateFormat.setLenient(false);

    binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat, false));
}
```

# Partie 4

# Validation de formulaire

- Interface `Validator`
- Exemple d'implémentation
- Mise en œuvre
- Annotation `@Valid`

# Interface Validator

Pour personnaliser la validation des données d'un formulaire, des implémentations de l'interface `Validator` doivent être fournies à Spring.

```
public interface Validator {

    boolean supports(Class aClass);

    void validate(Object object, Errors error

}
```

La méthode « `supports` » permet de spécifier sur quel Bean de formulaire peut être appliquée l'entité de validation.

La méthode « `validate` » contient l'implémentation de la validation.

# Exemple d'implémentation

```
public class PersonValidator implements Validator {

    public boolean supports(Class aClass) {
        return Person.class.isAssignableFrom(aClass);
    }

    public void validate(Object obj, Errors e) {
        ValidationUtils.rejectIfEmpty(e, "name", "name.empty");
        Person p = (Person) obj;
        if (p.getAge() < 0) {
            e.rejectValue("age", "negativevalue");
        } else if (p.getAge() > 110) {
            e.rejectValue("age", "too.darn.old");
        }
    }
}
```

1

2

1) les Beans de type « Person » pourront être vérifiés par ce validateur
2) toutes les erreurs sont ajoutées dans le conteneur de type « Errors »

# Mise en œuvre

Exemple de validation personnalisée des données dans une méthode de contrôleur :

```
@RequestMapping(method = RequestMethod.POST)
public String submitForm(@ModelAttribute Person person, BindingResult result) {

    (new PersonValidator()).validate(person, result);

    if (!result.hasErrors()) {
        [...]
        personManager.updatePerson(person);
    }
    return "userinfo";
}
```

1) la soumission du formulaire doit être de type POST
2) BindingResult étend la classe Errors et stocke le résultat des erreurs de conversion
3) instanciation et appel du validateur
4) si le conteneur d'erreurs est vide, on effectue les traitements métier

# Annotation @Valid (1/3)

Depuis Spring 3, l'annotation @Valid permet la validation implicite des données du formulaire par appel du validateur correspondant :

```
@RequestMapping(method = RequestMethod.POST)
public String submitForm(@Valid Person person) {

    [...]
    personManager.updatePerson(person);

  return "userinfo";
}
```

# Annotation @Valid (2/3)

Le validateur correspondant sera déclaré soit dans le contrôleur :

```
@InitBinder
protected void initBinder(WebDataBinder binder) {
    binder.setValidator(new PersonValidator());
}
```

# Annotation @Valid (3/3)

… soit globalement (pour tous les contrôleurs), dans le fichier de configuration XML de Spring :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    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/mvc
        http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">

    <mvc:annotation-driven validator="personValidator"/>

</beans>
```

# Partie 5

# Mécanisme de sélection de la vue

- Introduction
- Implémentations `ViewResolver`
- Exemples
- Chaînage de résolveurs

# Introduction

En sortie du traitement effectué par le contrôleur, un nom de vue « logique » est retourné.

Spring utilise deux interfaces pour trouver, à partir de ce nom de vue logique, la vue « physique » correspondante, dans la technologie de présentation choisie :

1) `ViewResolver` : fournit le mapping entre le nom de la vue logique et la vue physique.
2) `View` : transforme la requête dans l'une des technologies de vues (Freemarker, JSP, ...).

# Implémentations ViewResolver

| implémentation | description |
|---|---|
| `AbstractCachingViewResolver` | Classe de base qui met en cache le nom des vues une fois résolues. |
| `XmlViewResolver` | Décrit au cas par cas des vues utilisées dans un fichier XML. Par défaut `/WEB-INF/views.xml`. |
| `ResourceBundleViewResolver` | Décrit au cas par cas les vues utilisées dans un fichier de propriétés. Par défaut `views.properties`. |
| `UrlBasedViewResolver` | Les vues sont déterminées directement par les URL, sans configuration supplémentaire. |
| `InternalResourceViewResolver` | Sous-classe de `UrlBasedViewResolver`. Construit l'URI à partir de l'identifiant de la vue puis dirige les traitements vers d'autres ressources gérées par le conteneur de servlets. |
| `VelocityViewResolver / FreeMarkerViewResolver` | Sous-classes de `UrlBasedViewResolver` pour `VelocityView`, `FreeMarkerView`, et des `View` personnalisées. |

## Exemple UrlBasedViewResolver

Dans l'exemple suivant, le nom de vue logique « test » sera préfixé par « /WEB-INF/jsp/ » et suffixé par « .jsp » par le résolveur UrlBasedViewResolver.

Le nom physique de la vue retournée sera : /WEB-INF/jsp/test.jsp

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

# Exemple ResourceBundleViewResolver

Déclaration du résolveur :

```
<bean id="viewResolver"
class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
    <property name="basename" value="vues"/>
</bean>
```

Toutes les vues doivent ensuite être déclarées dans le fichier `vues.properties` :

```
user_list.class=org.springframework.web.servlet.view.RedirectView
user_list.url=users.action

admin_report.class=org.springframework.web.servlet.view.RedirectView
admin_report.url=admin.action
```

## Chaînage de résolveurs de vues

Plusieurs résolveurs de vues peuvent être déclarés dans le fichier de configuration.

La propriété « order » permet de définir l'ordre dans lequel Spring les utilisera séquentiellement pour déterminer les vues.

Cette configuration est utile par exemple lorsque des vues de type Excel ou PDF côtoient des pages web; les traitements de construction des vues sont alors différents.

# Partie 6

# Technologies de présentation

- JSP et JSTL
- Autres technologies

# JSP et JSTL

Spring MVC fournit une vue fondée sur JSP/JSTL, dirigeant les traitements de la requête vers une page JSP dont l'URI est construit à partir de l'identifiant de la vue. Il met a disposition:

- le contenu du modèle dans la requête
- les informations concernant le formulaire
- la gestion des éléments d'un formulaire avec le tag `form` (input, checkbox, ...)

## Autres technologies

Spring MVC fournit également un support pour les technologies suivantes :

- génération de documents (PDF, Excel, etc.) avec POI et Itext
- génération de rapports avec Jasper Reports
- génération de présentations fondées sur des templates (Velocity, FreeMarker)
- génération de présentations fondées sur les technologies XML (XSLT, ...)

# Partie 7

# Gestion des exceptions

- Introduction
- Interface HandlerExceptionResolver
- Implémentation SimpleMappingExceptionResolver
- Annotation @ExceptionHandler

# Introduction

Les exceptions sont par défaut propagées au conteneur de *servlets*.

Spring MVC propose trois solutions pour personnaliser le traitement des exceptions :

- définir son propre gestionnaire d'exception en implémentant l'interface `HandlerExceptionResolver`
- utiliser l'implémentation fournie : `SimpleMappingExceptionResolver`
- utiliser l'annotation `@ExceptionHandler`

# Interface HandlerExceptionResolver

Cette interface fournit la requête, la réponse, ainsi que l'exception survenue et le contrôleur dans lequel elle a eu lieu (handler).

```
public interface HandlerExceptionResolver {

    ModelAndView resolveException(
         HttpServletRequest request,
         HttpServletResponse response,
         Object handler,
         Exception ex);
}
```

# Implémentation SimpleMappingExceptionResolver

Cette implémentation fournie se configure en précisant, pour chaque type d'exception, la vue associée :

```
<bean id="exceptionResolver"
class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
  <property name="exceptionMappings">
    <props>
      <prop key="org.springframework.dao.DataAccessException">
        dataAccessFailure
      </prop>
      <prop key="org.springframework.transaction.TransactionException">
        dataAccessFailure
      </prop>
    </props>
  </property>
</bean>
```

L'exception est stockée dans la requête avec la clé « exception », ce qui la rend disponible pour un éventuel affichage.

# Annotation @ExceptionHandler

Cette interface fournit la requête, la réponse, ainsi que l'exception survenue et le contrôleur dans lequel elle a eu lieu (handler).

```
@Controller
public class SimpleController {

  @ExceptionHandler(IOException.class)
  public String handleIOException(IOException ex, HttpServletRequest request) {
    return ClassUtils.getShortName(ex.getClass());
  }

  [...]
}
```

Cette méthode sera appelée lorsqu'une exception de type « IOException » survient pendant les traitements du contrôleur.