Le framework Spring

Module 4 – Spring Web

Objectifs

- Les bases
- Le moteur de template Thymeleaf
- Les contextes d'exécution
- Les formulaires
- La validation des données
- L'internationalisation

Spring Web – Les bases

Objectifs

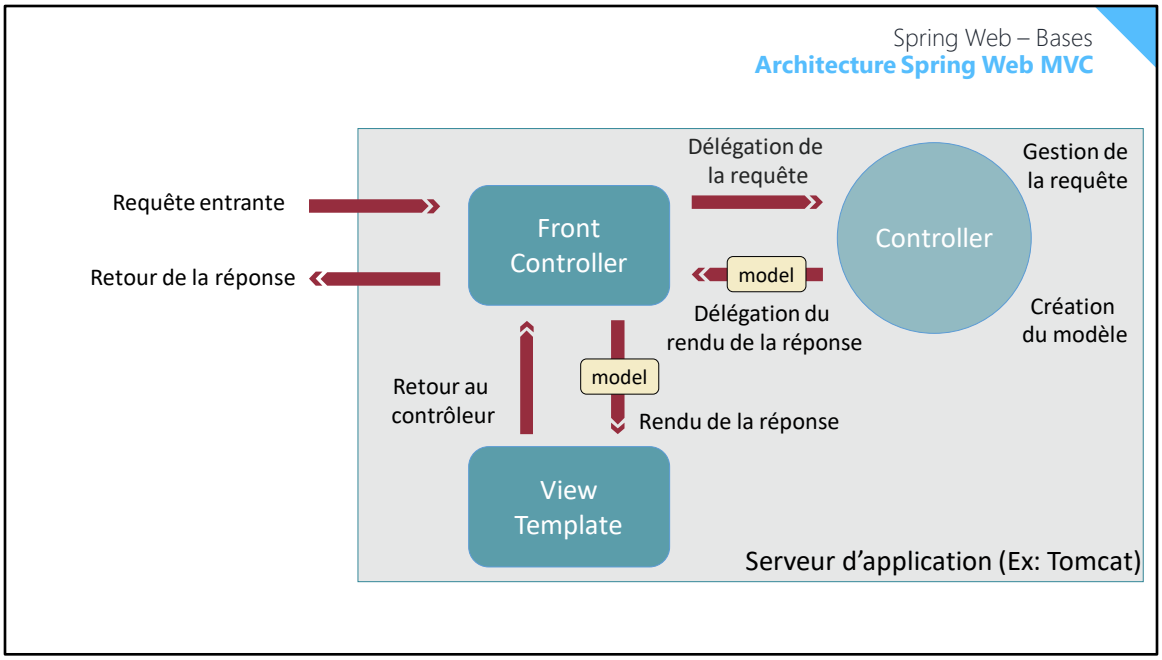
- Le design pattern Modèle Vue Contrôleur et l'architecture Spring MVC
- Créer un projet Spring web et Thymeleaf avec Spring Boot
- Créer un contrôleur avec l'annotation @Controller
- Déclencher un traitement java depuis une URL
- Déléguer le traitement d'une requête à une vue
- Lire des paramètres et les variables http
- Utiliser le modèle (Model, @ModelAttribute)
- Gérer une redirection 302

- MVC est un design pattern destiné aux interfaces graphiques
- Il est composé de trois parties :
  - Le contrôleur qui contient la logique applicative et gère les actions utilisateurs
  - Le modèle qui contient les données d'affichage
  - La vue qui définit la structure et l'apparence de l'IHM

L'architecture *Modèle/Vue/Contrôleur* (MVC) est une façon d'organiser une interface graphique d'un programme. Elle consiste à distinguer trois entités distinctes qui sont, le *modèle*, la *vue* et le *contrôleur* ayant chacun un rôle précis dans l'interface.

Le design pattern MVC se compose de trois modules :

- Un *modèle* représentant la structure logique sous-jacente des données dans une application logicielle, ainsi que la classe supérieure qui y est associée. Ce modèle d'objet ne contient aucune information sur l'interface utilisateur.
- Une *vue*, autrement dit un ensemble de classes représentant les éléments de l'interface utilisateur (tous ceux que l'utilisateur voit à l'écran et avec lesquels il peut interagir : boutons, boîtes de dialogue, etc.).
- Un *contrôleur* représentant les classes qui se connectent au modèle et à la vue, et servant à la communication entre les classes dans le modèle et la vue.

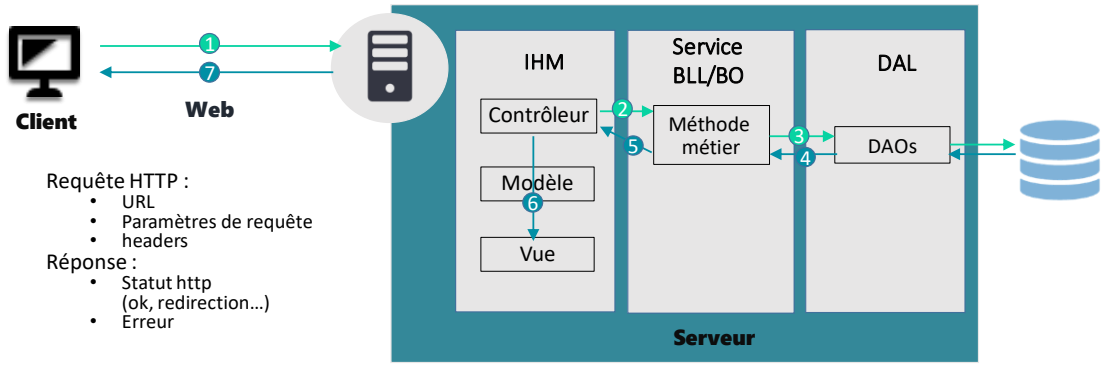


Le traitement des requêtes http par le serveur d'application s'organise autour d'un composant unique appelé Front Controller.

Le Front Controller nommé DispatcherServlet est une Servlet. Il hérite de HttpServlet défini par la norme JEE.

Voici les phases d'une requête avec Spring MVC :
1. Le Front controller est chargé de traiter les requêtes entrantes
2. Le front controller recherche le controller qui doit traiter la requête en fonction de l'url entrante
3. Une fois trouvé le controller va créer le modèle, traiter la requête puis déléguer le traitement d'affichage de la vue à un moteur de template défini au niveau du front controller

Architecture d'une application web



Voici l'architecture complète d'une application Web Java EE avec Spring MVC.
L'IHM (Interface Homme Machine) avec le pattern MVC décrit précédemment

- Ses contrôleurs vont transmettre ou récupérer les données des couches BLL et DAL pour créer ou mettre à jour le modèle

Ajout de Starters

- Pour le développement d'application Spring MVC :
  - spring-boot-starter-web
- Pour utiliser le moteur de template Thymeleaf :
  - Spring-boot-starter-thymeleaf
- Pour gérer le redéploiement automatique du serveur :
  - Spring-boot-devtools

```
dependencies {
  implementation 'org.springframework.boot:spring-boot-starter-web'
  implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
  developmentOnly 'org.springframework.boot:spring-boot-devtools'
}
```

build.gradle

Nous voulons maintenant créer une application web avec Spring boot et Gradle.
Pour rappel, Spring boot est un outils qui va créer le kit de démarrage utilisant les modules voulus (appelés starters) et le gestionnaire de dépendance souhaité (Maven ou Gradle)

Pour créer une application web, nous avons besoin du starter web et d'un moteur de template.

- Lemoteur de template est l'outils qui génère les pages web en liant les données traitées à des modèles écrits en html.
- Ces modèles sont appelées des vues.

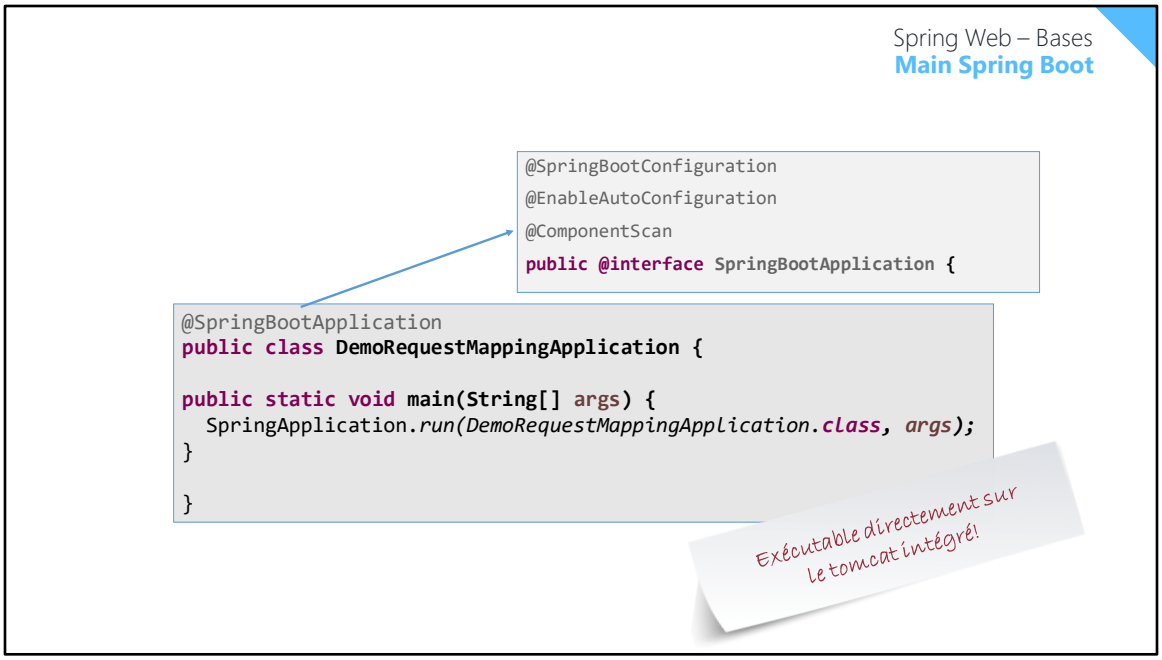
Plusieurs technologies de vues existent aujourd'hui. Parmi elles les Java Server Pages, et Thymeleaf.
La technologie Thymeleaf proposée par défaut dans Spring est plus récente et permet d'être utilisée en dehors du cadre de l'application web, notamment dans le formatage d'emails ou d'impressions.

- C'est la technologie que nous avons choisi dans ce cours.

La dépendance SpringBootDevTools utile pendant la phase de développement uniquement, est un outils qui permet le rechargement automatique du serveur

d’application lors de modification du code.

- Ce starter n’est pas obligatoire mais nous permettra de gagner notre temps si précieux.



https://docs.spring.io/spring-boot/docs/current/api/

@SpringBootApplication est l'équivalent de :

- @SpringBootConfiguration
- @EnableAutoConfiguration
- @ComponentScan

@SpringBootApplication Utilisé une seule fois dans l'application pour déclencher automatiquement l'ensemble des configurations de votre application lors de son exécution.

Elle déclenche les 3 autres annotations :

@SpringBootConfiguration → Précise qu'une classe annotée par elle peut remplacer une classe de configuration (@Configuration)

@EnableAutoConfiguration → Activez la configuration automatique du contexte d'application Spring. En déterminant et configurant les beans dont l'application à besoin.

Si des classes de configuration sont déclarées manuellement, elle les recherchera dans les sous packages de la classe.

@ComponentScan → Scannage dans les sous packages de cette classe pour trouver

l’ensemble des annotations liées à Spring.

- C'est un Bean Spring
  - Défini avec l'annotation @Controller
- Sert à traiter les requêtes HTTP
- Permet l'interaction avec le modèle
- Délègue le traitement de la requête à une vue ou renvoie la réponse
- Les requêtes peuvent être mappées au niveau de la classe ou des méthodes
  - @RequestMapping
  - @GetMapping, @PostMapping, ...

C'est un bean Spring

Défini avec l'annotation @Controller

Il doit être placé dans l'arborescence de la classe d'application

Sert à traiter les requêtes HTTP

Permet l'interaction avec le modèle

Délègue le traitement de la requête à une vue ou renvoie la réponse

Si la méthode retourne le nom d'une vue, c'est une délégation de traitement (forward)

Si la chaine retournée est préfixée par redirect:, c'est une redirection (statut http 301)

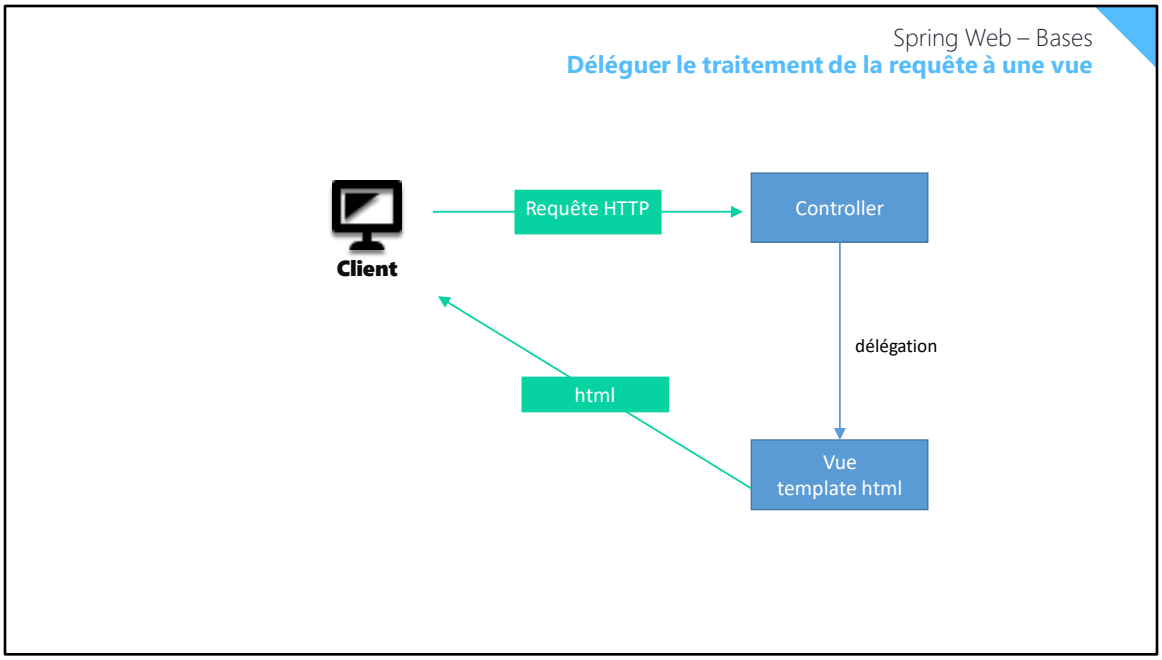
Les requêtes peuvent être mappées au niveau de la classe ou des méthodes

- @RequestMapping,
    - Peut se placer sur la classe annotée @Controller pour préciser une URL racine pour toutes les méthodes de la classe
    - peut être déclarée sur les méthodes Java pour gérer le mapping entre elles et une URL
        - Il faut préciser dans ses paramètres : l'url et la méthode du protocole http (get, post, put, delete, …)
        - Exemple : @RequestMapping(value = "/get/{id}", method = RequestMethod.GET)

- Pour le mapping des méthodes de la classe, il y a une nouvelle approche.
    - C'est une solution plus directe et plus simple : @GetMapping ou @PostMapping, … (toutes les méthodes du protocole http sont représentées)
    - Le nom de la méthode du protocole http ; est précisé par l'annotation et il suffit d'intégrer l'url en paramètre
    - Exemple : @GetMapping(value = "/get/{id}")

- Les 2 solutions sont viables et peuvent être utilisées.

Nous verrons cela en détail dans la démonstration



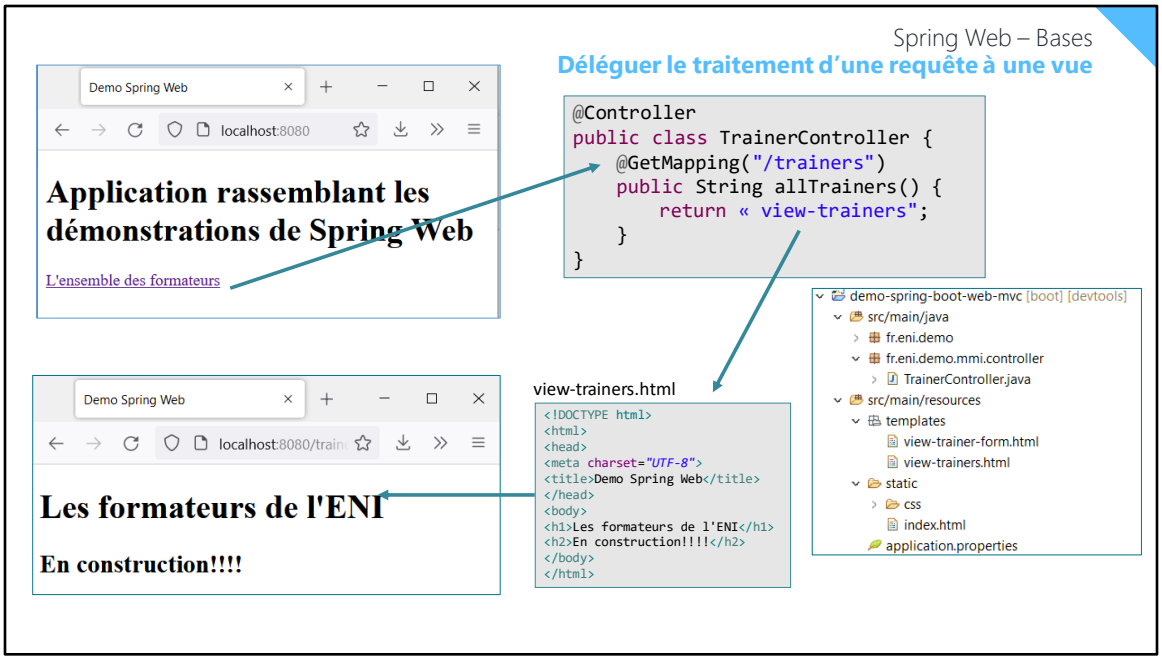
Spring considère que les méthodes des contrôleurs mappés à une url retournent le nom d'une vue.
Par défaut, le fichier définissant la vue sera : nom de la vue + «.html »

Le suffixe peut être redéfini

Les vues (templates html) sont placées dans le répertoire templates
Les pages et éléments statiques (css/js/images) sont placés dans le répertoire static

Voyons dans une démonstration la mise en

pratique.



Voici un exemple de délégation.
Sur la page d'index; un lien hypertexte « L'ensemble des formateurs » doit déclarer l'url : « trainers ».
Ainsi le DispatcherServlet de Spring cherchera le contrôleur capable d'y répondre dans notre cas :

- TrainerController
- Et il activera la méthode allTrainers associée.

Cette méthode retourne une chaîne de caractères qui est l'alias de la vue associée.
Dans notre exemple view-trainers.html.
Le dispatcherServlet retournera ainsi la vue souhaitée.

Nous allons mettre en pratique cet exemple dans la démonstration suivante.

Le mapping des requêtes

- Spring permet d'utiliser 2 stratégies pour le passage de valeurs dans requête HTTP
  - Paramètres de requêtes HTTP (@RequestParam)
  - Variables de l'URI (@PathVariable)

- Comprendre la différence d'utilisation :

http://localhost:8080/trainers/10/trainings?tag=Spring

  - 10 est une variable, récupérable avec @PathVariable
  - tag=Spring est un paramètre, récupérable avec @RequestParam

Spring permet d'utiliser 2 stratégies pour le passage de valeurs dans requête HTTP

- Paramètres de requêtes HTTP (@RequestParam)
- Variables de l'URI (@PathVariable)

@RequestParam et @PathVariable peuvent toutes les deux être utilisées pour extraire des valeurs de la requête, mais elles sont un peu différentes.

- @RequestParams extrait les valeurs de la chaîne de requête
- @PathVariables extrait les valeurs du chemin URI

D'autre part @PathVariable extrait des valeurs du chemin URI, il n'est pas encodé. En revanche, @RequestParam est encodé.

- @RequestParam : http://localhost:8080/spring-mvc-basics/foos?id=ab+c
  - La donnée est encodée → ID: ab c
- @PathVariable : http://localhost:8080/spring-mvc-basics/foos/ab+c
  - La donnée est directement → ID: ab+c

Comprendre la différence d'utilisation :
http://localhost:8080/trainers/10/trainings?tag=Spring

    - Dans cette URL;
    - 10 est une variable, récupérable avec @PathVariable
    - tag=Spring est un paramètre, récupérable avec @RequestParam

- @PathVariable est souvent utilisé pour l'architecture REST. Spring REST les exploitent nous le verrons un peu plus tard.

Lecture des paramètres HTTP

- Le paramètre peut être :
  - Dans l'URL directement, derrière le ? `<a href="trainers/detail?email=abaille@campus-eni.fr">`
  - Au travers d'un formulaire (attribut « name » des champs) `<input type="text" name="email"/>`
- @RequestParam sur le paramètre de la méthode du contrôleur permet de le récupérer

```
public String detailTrainer(
@RequestParam(name = "email", required = false, defaultValue = "coach@campus-eni.fr") String emailTrainer) {
```

- Le membre d'annotation « required » permet d'indiquer si le paramètre http est optionnel ou non
  - Quand le paramètre est optionnel on peut définir une valeur par défaut

Le paramètre peut être :

Dans l'URL directement, derrière le ?

Au travers d'un formulaire (attribut « name » des champs)

@RequestParam sur le paramètre de la méthode du contrôleur permet de le récupérer

Soit en précisant le membre « name » de l'annotation

Soit le paramètre a le même nom que l'attribut « name » de l'url

Le membre d'annotation « required » permet d'indiquer si le paramètre http est optionnel ou non

Quand le paramètre est optionnel on peut définir une valeur par défaut

Lecture des variables

- La variable est déclarée dans l'URI directement
  - Sans encodage

```
<a href="trainers/detail/variable/sgobin@campus-eni.fr">
```

- Côté contrôleur :
  - Définir la variable à l'emplacement attendu dans l'URL avec la syntaxe {nomVariable}
  - Injecter la valeur avec l'annotation @PathVariable

```
@GetMapping("/detail/variable/{email}")
public String detailTrainer2(@PathVariable(name = "email") String emailTrainer) {
```

La variable est déclarée dans l'URI directement

Sans encodage

Côté contrôleur :

Définir la variable à l'emplacement attendu dans l'URL avec la syntaxe {nomVariable}

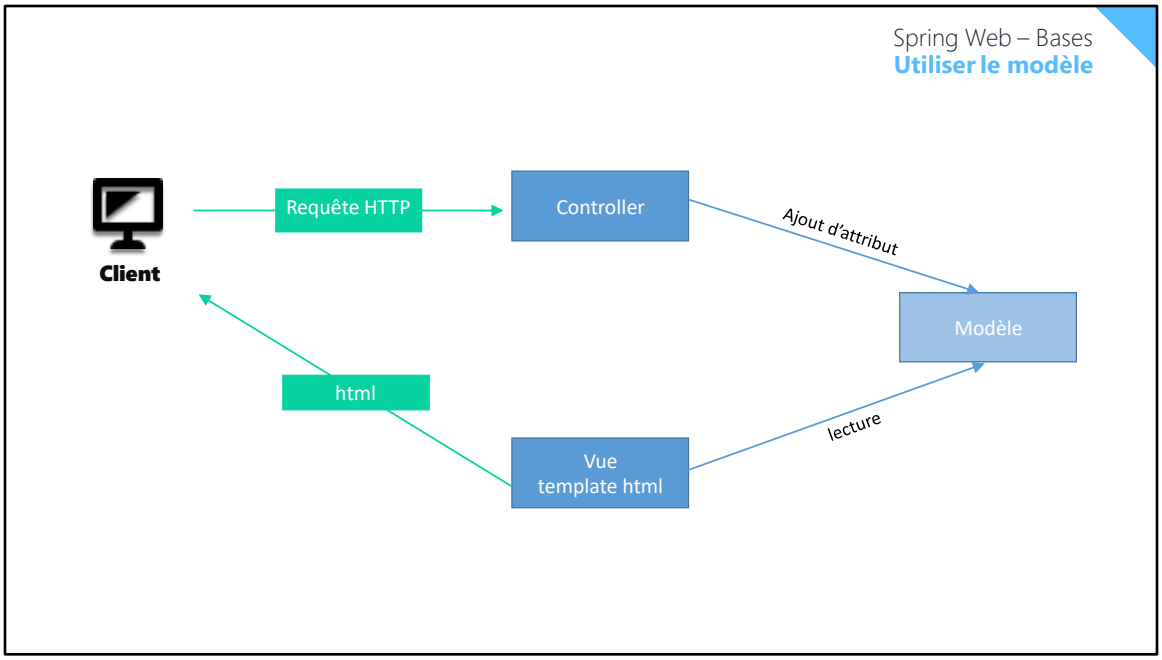
Injecter la valeur avec l'annotation @PathVariable

Il est possible de mettre la variable en optionnelle, dans ce cas ajouter le membre « required = false »

- Il faudra compléter la liste des url possibles avec celle sans la variable
- Par défaut, la variable vaudra nulle. Il n'y a pas de membre à l'annotation pour ajouter une valeur par défaut. Il faudra le faire dans le code.

Nous allons voir ces passages de paramètres et de variables en détail au travers de la démonstration suivante.

Les paramètres et variables http



Nous venons de voir le passage de valeurs dans la requête.
Il serait intéressant de voir comment transférer des données vers la vue pour mettre à jour l'affichage.
Pour cela dans les contrôleurs, nous pouvons gérer des attributs et les utiliser pour manipuler les données de la vue.
En Spring, ces attributs sont manipulés au travers de la classe `org.springframework.ui.Model`

Ainsi,

- le contrôleur crée des données dans le modèle
- la vue lit le modèle pour l'afficher

Pour le moment, nous allons voir la manipulation en lecture. Quand nous détaillerons Thymeleaf, nous verrons comment utiliser les données de la vue

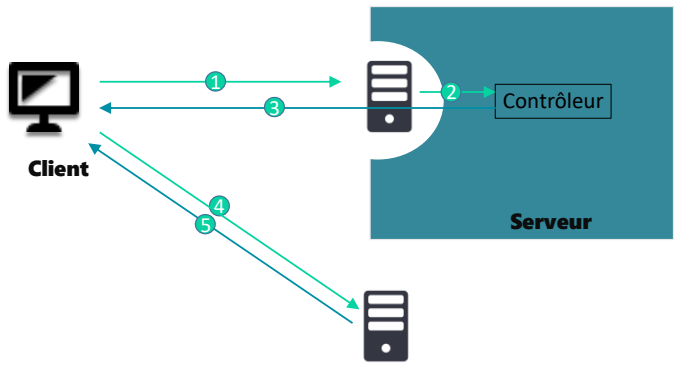
Utiliser le modèle

- Côté contrôleur :
  - Injecter le modèle dans la méthode en ajoutant un paramètre de type `org.springframework.ui.Model`
  - `Les données sont stockées dans le modèle sous forme d'attributs ( clé – valeur )`
- Côté vue :
  - Lire les données du modèle suivant la technologie de vue choisie
  - Nous utiliserons Thymeleaf

Voyons dans le détail le code correspond dans la démonstration qui suit.

Utiliser le modèle

Spring Web – Bases
Redirection



- Au lieu de déléguer le traitement d'une requête à une vue, le contrôleur peut renvoyer directement une réponse de type redirection (statut HTTP 302)
  - Change l'url et la requête HTTP courante sur le client
  - Dirige l'utilisateur à l'extérieur de la Webapp
  - Inconvénient : 2 allers-retours sont nécessaires pour afficher la page

Au lieu de déléguer le traitement d'une requête à une vue, le contrôleur peut renvoyer directement une réponse de type redirection (statut HTTP 302)
[301 – redirection permanente]
[302 – redirection temporaire]

Cela permet de changer l'url et la requête HTTP courante sur le client
Cela permet de diriger l'utilisateur à l'extérieur de la webapp
Cela permet de rediriger vers un autre contrôleur pour activer un comportement
Inconvénient : 2 allers-retours sont

nécessaires pour afficher la page

1- Requête envoyée au serveur d'application
2- Délégation à un Contrôleur
3- Renvoi du statut 301 et de l'url à afficher
4- Le navigateur client déclenche automatiquement une requête HTTP Get vers l'url demandée
5- Le serveur visé renvoie la page demandée

Note : Utile quand on veut éviter les soumissions multiples de formulaire

Redirection

- Pour faire une redirection, il faut que la méthode mappée renvoie le mot clé redirect: suivi de l'url cible
- Syntaxe : « redirect:/url_de_redirection »

```
return "redirect:/index.html";
```

Mettons en pratique dans une démonstration.

Redirection

Introduction au moteur de template Thymeleaf

Objectifs

- Qu'est ce que Thymeleaf
- Comment intégrer Thymeleaf dans un projet
- Créer une vue Thymeleaf
- Intégrer les instructions de Thymeleaf

Qu'est ce que Thymeleaf ?

- Un moteur de modèle (template engine)
- Permet d'intégrer des données dans un modèle de document pour générer :
  - page web, email, fichier csv, pdf ou autre
- Adapté aux applications web :
  - Alternative aux JSP
  - Indépendant du framework Spring
  - Ne fait qu'ajouter ses propres attributs html, ce qui permet de visualiser le rendu d'une page sans exécuter l'application

Comment intégrer Thymeleaf dans un projet

- Nécessite la dépendance Thymeleaf
- Pour Gradle ajouter la dépendance comme suit :

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
    …
}
```

Comment utiliser Thymeleaf dans une vue

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
…
</head>
<body>
…
</body>
</html>
```

template.html

Il suffit d'ajouter le namespace Thymeleaf !

Le fichier est suffixé par l'extension .html et doit être placé dans le dossier templates

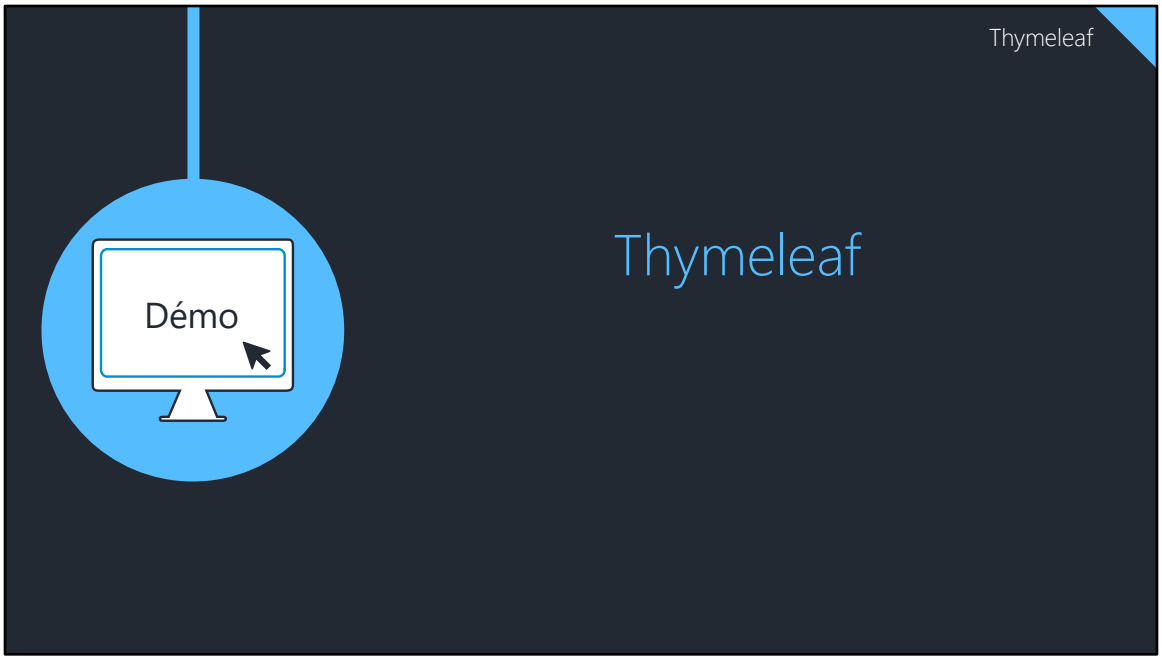
Intégrer les instructions Thymeleaf dans une vue

- La plupart des instructions Thymeleaf sont ajoutées en tant qu'attribut dans les balises et utilisent le préfixe défini par le namespace (th)
  - Pour que les attributs soient valides pour HTML5 : il faut les préfixer par data- et utiliser le séparateur « - » au lieu du « : »
- Les instructions peuvent utiliser le Spring Expression language (Spel)
- Le résultat de l'expression est affecté au contenu de la balise
- Pour plus d'informations : https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html

- Les expressions langages vont permettre de récupérer de l'information à l'extérieur de la vue et l'incorporer dans la page web générée :

Objectif	**Syntaxe**	**Exemple(s)**
Expressions variables (accès aux données du modèle, beans…)	${ … }	${'nom : ' + personne.nom} ${@myBean.doSomething()}
URL	@{ … }	@{/order/details(orderId=${o.id})}
Messages	#{ … }	#{home.welcome}
Expressions de sélection – permet un accès direct aux membres d'un objet	*{ … }	<body th:object="${order}"> 99 (Est equivalent à ${order.id})

Faisons plusieurs exemples pour manipuler les instructions de Thymeleaf et de Spring EL; dans la démonstration suivante



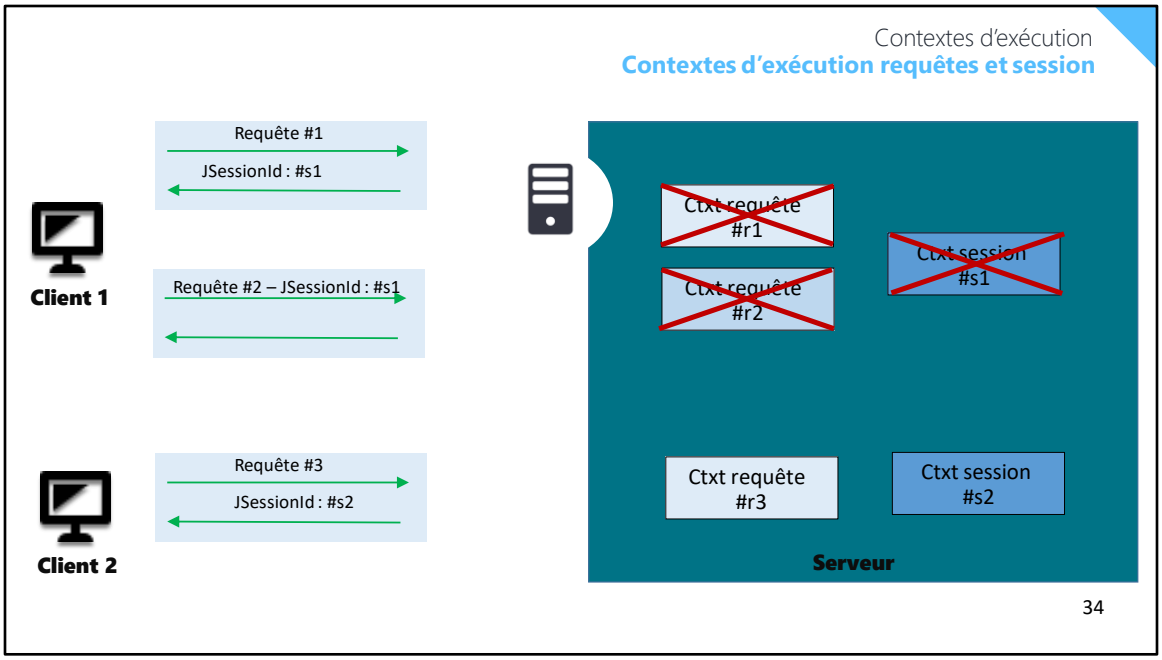
Remarques :

- Thymeleaf ne fait qu'ajouter ses balises, ce qui permet l'affichage des pages dans un navigateur sans l'exécution de l'application sur le serveur d'application.

- Cependant, toutes les balises Thymeleaf apportent les informations dynamiques. Pour cela, il faut le serveur d'application.

TP – Filmothèque - Partie 02

Contextes d'exécution



Le protocole HTTP étant sans état, les données portées par une requête sont perdues à la fin du traitement de celle-ci.
Sur l'exemple,

- le client 1 envoie une requête 1 au serveur,
- le serveur d'application Java crée un espace mémoire pour traiter cette requête.
- Les données contenues dans ce contexte mémoire de requêtes vont disparaitre quand la réponse sera renvoyée,
- sauf si une session a été créée.
    - Dans ce cas, pour l'ensemble des requêtes d'un même client, il sera possible de conserver les informations dans sa session.

Le contexte de session vivra tant que le client envoie des requêtes avant le timeout de session.
Et/ou tant que le client ne ferme pas son navigateur.

Chaque client peut avoir son espace de session, ce qui permet de gérer des données spécifiques à l'utilisateur.

Contextes d'exécution
Gestion des données en session

- @SessionAttributes
  - Permet de définir une liste d'attribut qui doivent être gérés en session
  - Injecter cette liste sur tous les contrôleurs qui les affichent
- @SessionScope
  - Ajouté à la définition d'un bean, et dans le contexte d'une Webapp
- HttpSession
  - Utiliser la gestion des sessions de Http
- ATTENTION :
  - Spring charge le contexte en Session au chargement de la vue
  - HttpSession, ajoute immédiate l'attribut

Mettre des données en session, évitera le rechargement continuel de données.
Utile pour des listes de données qui ne varient pas ou peu dans un formulaire par exemple.
Utile pour gérer un utilisateur connecté.

@SessionAttributes

- Permet de définir une liste d'attribut qui doivent être gérés en session
- @SessionAttributes se place sur la classe du contrôleur qui veut manipuler la donnée
- Si vous voulez manipuler cette donnée depuis une méthode model.getAttribute
- Attention, Spring charge ces valeurs dans le contexte de session au chargement de la vue
  - Suppression des attributs par `status.setComplete();` et un redirect (pour réaliser un refresh complet)

- Attention, pour que Spring permettent sur chaque vue d'afficher les attributs en session, il faut que cette annotation soit placer sur tous les contrôleurs qui les manipulent

@SessionScope

- Ajouté à la définition d'un bean (@Bean), et dans le contexte d'une webapp (@Configuration)
- Permet d'obtenir un bean qui a comme vie la durée de la session
- Permettra de gérer des configurations spécifiques

HttpSession

- Utiliser la gestion des sessions de Http
- Passer en paramètre des méthodes du contrôleur
- Suppression de la session avec la méthode invalidate()

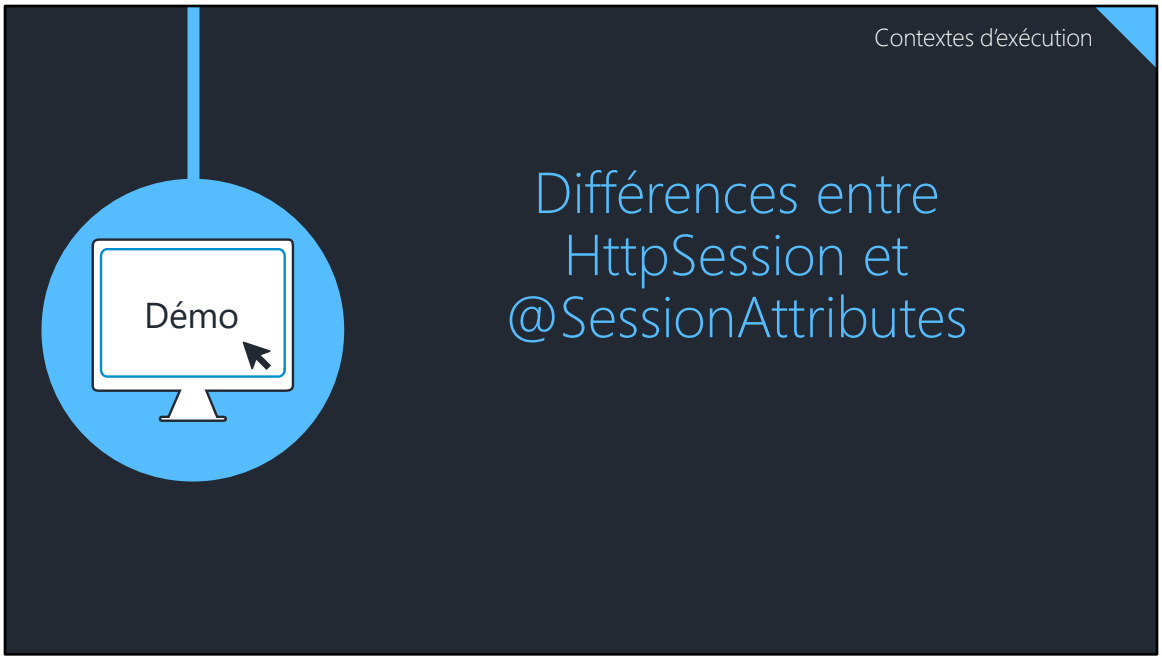
ATTENTION :

Spring charge le contexte en Session au chargement de la vue HttpSession, ajoute immédiate l'attribut

Contexte de session par Spring

Voyons dans le détail la gestion de session par Spring avec

`@SessionAttributes`



Voyons maintenant les différences entre HttpSession et @SessionAttributes

Contextes d'exécution
Contexte application

- Il est possible de manipuler le contexte de l'application
- ServletContext
  - Injecter dans le contrôleur
- @ApplicationScope
  - Ajouté à la définition d'un bean dans une classe de configuration

Il est possible de manipuler le contexte de l'application
ServletContext
Injecter dans le contrôleur
@ApplicationScope
Ajouté à la définition d'un bean dans une classe de configuration
Ce bean est accessible sur toute l'application.

Dans ce cas, l'information injectée est partagée entre tous les utilisateurs de l'application.
Attention, penser que ces données sont chargées sans limite de durée.

TP – Filmothèque - Partie 03

Formulaires

@ModelAttribute : 2 manières de manipuler le modèle

1) Porté par un paramètre de méthode mappée, @ModelAttribute permet d'injecter un unique attribut du modèle
   - Plutôt que tout le modèle (Model)
2) Porté par une méthode, il permet d'initialiser un attribut du modèle avant l'appel de la méthode mappée

1) Porté par un paramètre de méthode mappée, @ModelAttribute permet d'injecter un unique attribut du modèle
   - Plutôt que tout le modèle (Model)
   - Il est utilisé pour la gestion des formulaires

2) Porté par une méthode, il permet d'initialiser un attribut du modèle avant l'appel de la méthode mappée

Notes :
- La méthode annotée sera déclenchée avant chaque appel de méthode mappée, quelque soit la méthode. Autrement dit, sur chaque appel de requête
(Cela équivaut à mettre un attribut en request.)
- L'annotation doit indiquer le nom de l'attribut, la valeur de l'attribut sera le résultat de la méthode

- Gestion des données d'un formulaire directement à partir d'un objet
- Thymeleaf propose
  - l'instruction data-th-object pour référencer un objet du modèle
  - L'instruction data-th-field permet de faire référence aux membres de l'instance courante
- Côté contrôleur, il y a 2 choses à faire :
  - Initialiser les données du formulaire
  - Récupérer les données saisies par l'utilisateur

L'objectif est de pouvoir initialiser et récupérer les données d'un formulaire directement à partir d'un objet.

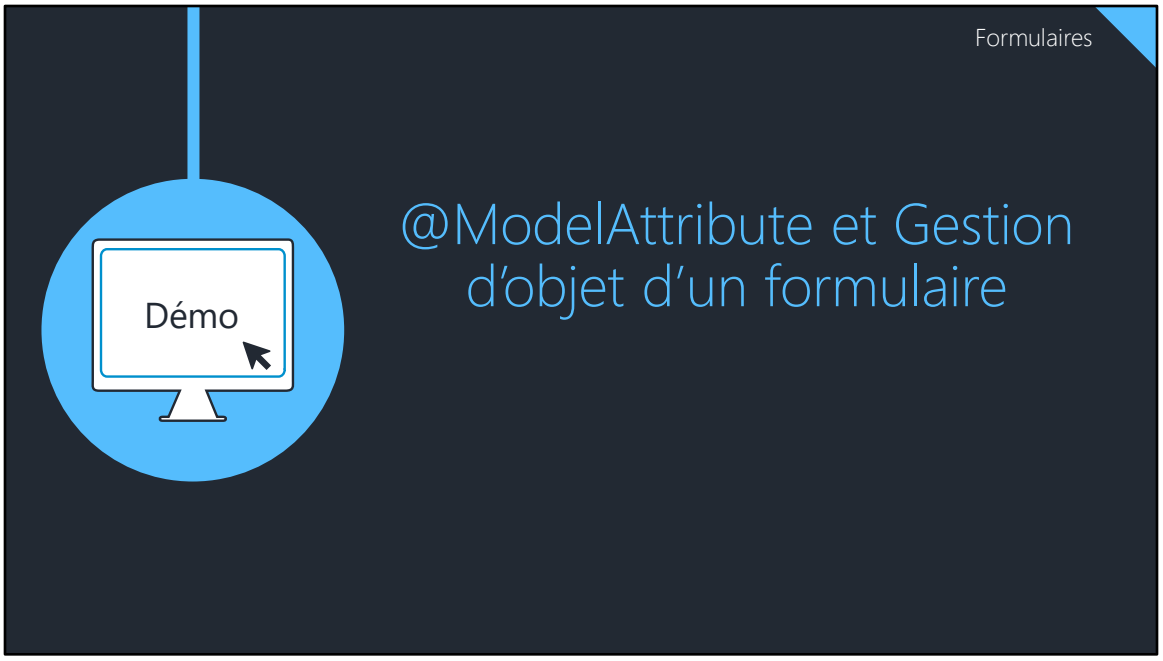
Côté vue, Thymeleaf propose

- l'instruction data-th-object pour référencer un objet du modèle
- L'instruction data-th-field permet de faire référence aux membres de l'instance courante (traduit en html par l'attribut name).

Côté contrôleur, il y a 2 choses à faire :

Initialiser les données du formulaire – créer l'instance de l'objet représenté par le formulaire et l'injecter dans le modèle

Récupérer les données saisies par l'utilisateur – utiliser @ModelAttribute pour récupérer l'instance du formulaire du modèle



Mettons en place ces éléments dans une démonstration.

Gérer une association avec un Converter

Les objets métiers ont souvent des associations entre eux.

- Côté contrôleur, cela impose un objet de formulaire plus complexe
- Côté vue, une liste de données à afficher en continue
- Exemple : Nos formateurs et leurs cours
  - Actuellement séparation de la mise à jour (attributs directs et association)

- Sans les Converter, il n'est pas possible de gérer le tout en un objet de formulaire
  - Ni d'afficher facilement la liste des données d'origine

Les objets métiers ont souvent des associations entre eux.
Côté contrôleur, cela impose un objet de formulaire plus complexe

Côté vue ;
Pour la gestion du formulaire cela veut dire parfois,

- une liste de données à afficher en continue
- Et de manière répétée (comme les cours pour nos formateurs)

Exemple : Nos formateurs et leurs cours

- Pour le moment, nous avons vu comment mettre à jour les données séparément
- D'un côté un formulaire qui permet de mettre à jour les attributs directs (nom, prénom, l'email)
- De l'autre la mise à jour de la liste des cours

Sans les Converter, il n'est pas possible de gérer le tout en un objet de formulaire

- Ni d'afficher facilement la liste des données d'origine

Converter, permet de gérer l'affichage et l'injection de l'association

- Définir un bean de Spring qui implémente

  Converter<Class<?> sourceType, Class<?> targetType>

- Redéfinir la méthode

  <T> T convert(Object source, Class<T> targetType);

Spring l'appel automatiquement

- Un attribut de Model qui correspond au type
- Ou sur un bean qui retourne des objets de ce type

Converter, permet de gérer l'affichage et l'injection de l'association
Mise en place :

- Définir un bean de Spring qui implémente

  Converter<Class<?> sourceType, Class<?> targetType>

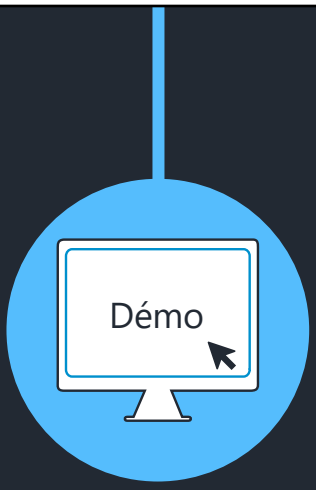
- Redéfinir la méthode

  <T> T convert(Object source, Class<T> targetType);

  - Cette méthode permet à l'affichage et au contrôleur de savoir comment récupérer une des informations

Spring l'appel automatiquement

- Un attribut de Model qui correspond au type
- Ou sur un bean qui retourne des objets de ce type



Utiliser un Converter

Mettons en place un Converter pour la création d'un formateur avec la liste de ses cours associés.

TP – Filmothèque - Partie 04

Validation des données

Validation des données

- Pourquoi valider les données ?
  - Pour l'intégrité des données
    - Les données reçues doivent correspondre aux format attendu pour permettre leur traitement et leur stockage
  - Pour la sécurité
    - Exemple : Commander un nombre d'article négatif pourrait permettre de créditer son solde bancaire !

Exemples de contraintes proposées

Contrainte	Signification (sur l'élément)
@AssertTrue	Doit être à true
@AssertFalse	Doit être à false
@Min, @DecimalMin	Doit être supérieur à …
@Max, @DecimalMax	Doit être inférieur à …
@Digits	Définit le nombre de chiffres
@Size	Doit être entre deux tailles
@Null	Doit être nul
@NotNull	Doit être non nul
@NotEmpty	Ne peut pas être nul ni vide
@NotBlank	Ne peut pas être nul ni vide après trim()

Contrainte	Signification (sur l'élément)
@Pattern	Doit respecter une RegExp
@Email	Possède le format email
@Past	Doit être dans le passé
@Future	Doit être dans le futur

Ces contraintes font partie de Spring

Exemples de contraintes proposées

- Quelques contraintes fournies spécifiquement par Hibernate Validator :

Contrainte	Signification (sur l'élément)
@CreditCardNumber	Représente un numéro de carte de crédit
@URL	Représente une URL valide
@Range	Doit être dans l'intervalle

Pour ces contraintes, il faut ajouter la librairie hibernate-validator

Validation des données

- Ajouter le starter : spring-boot-starter-validation
- Placer les annotations sur les objets métiers
- Déclencher la validation des données dans le Contrôleur :
  - Appel de méthode validant le formulaire : @Valid
  - Ajout du paramètre de type BindingResult
    - Ce paramètre doit suivre le paramètre annoté @Valid
- Affichage des erreurs sur la vue
  - Soit dans un bloc général
  - Soit sur chaque champ

Ajouter le starter : spring-boot-starter-validation dans build.gradle :
implementation 'org.springframework.boot:spring-boot-starter-validation'

Placer les annotations sur les objets métiers.

Déclencher la validation des données dans le Contrôleur :
- Appel de méthode validant le formulaire : @Valid
- Ajout du paramètre de type BindingResult
  - Ce paramètre doit suivre le paramètre annoté @Valid

Affichage des erreurs sur la vue
- Soit dans un bloc général
  - → pour savoir s'il y a des erreurs
  - Exemple :

```
<div style="color:red" data-th-
if="${#fields.hasErrors('*')}" >
            <p>Veuillez vérifier vos champs</p>
</div>
```

→ Parcourir les erreurs selon les champs :

Exemple :

```
<ul>
<li data-th-each="erreur:
${#fields.errors('lastName')}" data-th-
text="${error}"></li>
</ul>
```

Soit sur chaque champ → exemple

```
<div class="cards" data-th-
if="${#fields.hasErrors('lstCourses')}">
<span data-th-
errors="*{lstCourses}"></span>
</div>
```

- Les messages d'erreurs sont déterminés comme suit :
  - Message d'erreur défini par défaut par l'annotation
  - Attribut de l'annotation
    - Exemple `@NotBlank(message = "Le prenom est obligatoire")`
  - Dans la vue et en relation avec les **fichiers propriétés**
- Le message le plus spécifique est pris en compte

Les messages d'erreurs sont déterminés comme suit :

Message d'erreur défini par défaut par l'annotation

Exemple @NotBlank => Ne doit pas être vide

La langue utilisée est celle demandée par le navigateur (négociation de contenu)

Attribut de l'annotation

Exemple `@NotBlank(message = "Le prenom est obligatoire")`

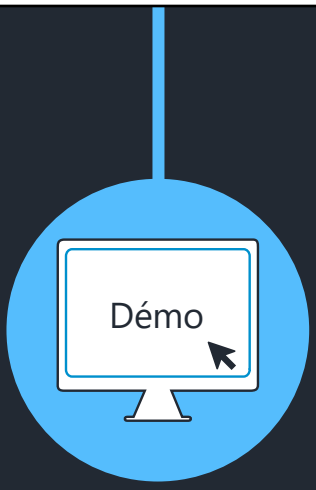
Dans la vue et en relation avec les **fichiers propriétés**

Exemple :

Fichier messages.properties :

```
NotBlank=Le champ est obligatoire
NotBlank.member.lastName=Le champ prénom est obligatoire
```

Le message le plus spécifique est pris en compte



Validation d'un formulaire

Mettons la validation des données en place dans notre démonstration.

TP – Filmothèque - Partie 05

Internationalisation

Mise en place de l'internationalisation sur les messages

- Ajout de fichier de properties par langues
  - messages.properties, messages_fr.properties, …
- Dans la configuration générale : application.properties
  - Ajout de l'encodage en UTF-8
  - `spring.messages.encoding=UTF-8`
- Au niveau des vues, utilisation de
  - `data-th-text="#{clef_dans_messages}"`

Gestion des langues

- Changement de langues :
  - Définition d'une classe de configuration implémentant WevMvcConfigurer et annotée @Configuration
  - Définir un bean LocaleResolver pour gérer la langue par défaut
  - Définir un bean LocaleChangeInceptor; pour intercepter le paramètre de changement de langue (exemple : language)
  - Redéfinir la méthode addInterceptors pour enregistrer le bean précédent
  - Sur les vues, ajouter aux URL le paramètre de langue et la Locale de la langue (exemple : ?language=en)

Mettons en place l'internationalisation dans notre démonstration.

TP – Filmothèque - Partie 06