



**SPRING** |  **AJC**



# SPRING WEB MVC

Introduction à  
Spring Web MVC

# SOMMAIRE

Présentation de Spring MVC

Configuration

Le Controller

Vues composites

Data Binding

Validation

Internationalisation

# PRÉSENTATION DE SPRING MVC

## Une Servlet principale : DispatcherServlet

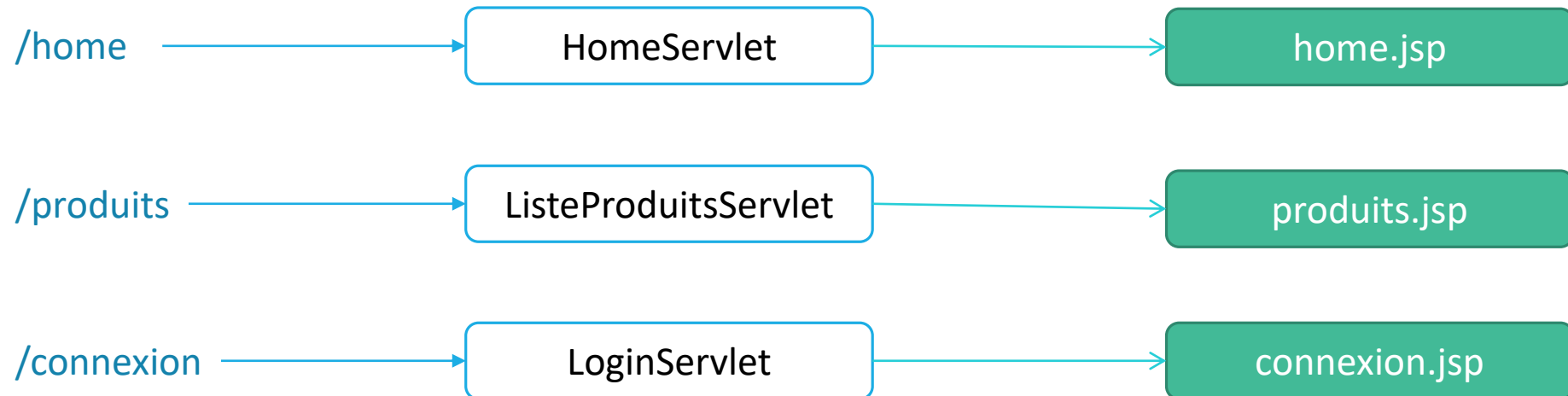
- Délègue les requêtes à des contrôleurs (classes annotées @Controller)
  - Selon le point d'accès (la ressource URL)

## Un controller

- Fabrique un modèle sous la forme d'une Map qui contient les éléments de la réponse
  - Clé / Valeur
- Utilise une View pour afficher la vue (la page HTML)

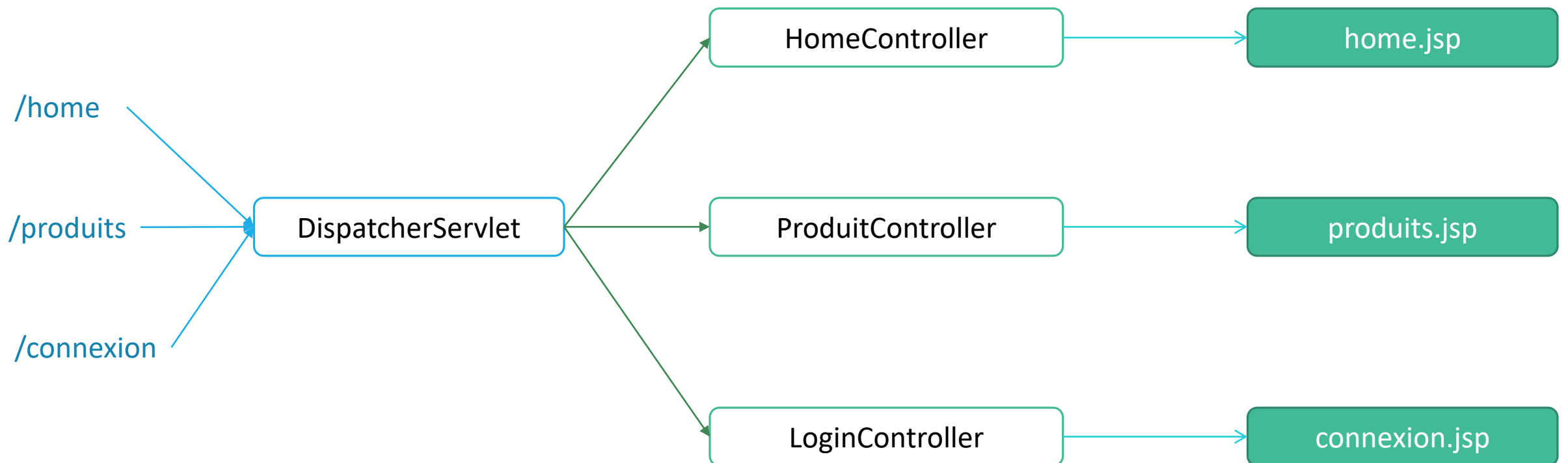
# PRÉSENTATION DE SPRING MVC

JEE Servlets classiques



# PRÉSENTATION DE SPRING MVC

Avec Spring MVC



# PRÉSENTATION DE SPRING MVC

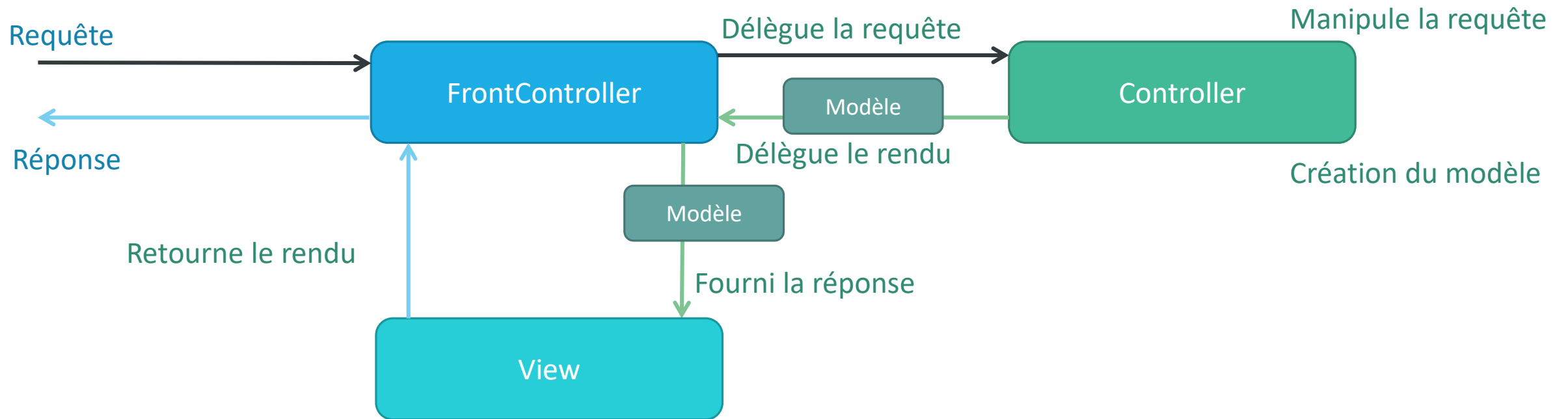
La Servlet DispatcherServlet est mappée sur toutes les ressources

- Par exemple « / »
- On l'appelle « FrontController »

Il n'y a plus de Servlet JEE

- Mais si elles existent, leur mapping prend le pas sur le mapping des @Controller !

# TRAITEMENT D'UNE REQUÊTE





# CONFIGURATION (FRONT CONTROLLER)

Déclaration de la Servlet unique dans le web.xml

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

  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/dispatcher-context.xml</param-value>
  </init-param>

  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>dispatcher</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>
```

DispatcherServlet utilise son propre fichier de configuration.  
Si vous définissez un fichier unique (application-context.xml)  
Vous aurez 2 instances que chaque bean Spring !

# CONFIGURATION (FRONT CONTROLLER)

## Configuration XML avec configuration Spring par classe

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

  <init-param>
    <param-name>contextClass</param-name>
    <param-value>org.springframework.web.context.support.AnnotationConfigWebApplicationContext</param-value>
  </init-param>

  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>fr.formation.config.WebConfig</param-value>
  </init-param>

  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>dispatcher</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>
```

# CONFIGURATION (FRONT CONTROLLER)

Si on veut ajouter la configuration générale Spring (JPA par exemple)

- Ne pas oublier de charger le contexte de Spring

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

- En plus du chargement de la configuration de Spring dans ce fichier **web.xml**

# CONFIGURATION (FRONT CONTROLLER)

Dans le fichier de configuration de DispatcherServlet

- ➔ dispatcher-context.xml
- Activer le contexte de Spring pour déléguer les requêtes aux contrôleurs
  - Cette balise crée deux beans
    - DefaultAnnotationHandlerMapping
    - AnnotationMethodHandlerAdapter
- Configuration XML

```
<mvc:annotation-driven />
```

- Configuration par classe

```
@EnableWebMvc
```

# CONFIGURATION (FRONT CONTROLLER)

DispatcherServlet est mappée sur toutes les ressources ...

- Comment accéder aux ressources CSS, JS, Images, ... ?
- Comment distribuer une ressource statique ?

Utilisation de la balise `mvc:resources` prévue à cet effet

- Dans la configuration de DispatcherServlet (dispatcher-context.xml)

```
<mvc:resources mapping="/css/**" location="/css/" />  
<mvc:resources mapping="/images/**" location="/images/" />
```

# CONFIGURATION (FRONT CONTROLLER)

En configuration par classe

- La classe de configuration doit implémenter `WebMvcConfigurer`
- Vous devez surcharger la méthode `addResourceHandlers()`

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/css/**").addResourceLocations("/css/");
        registry.addResourceHandler("/images/**").addResourceLocations("/images/");
    }
}
```

# CONFIGURATION (CONTROLLERS)

Annotation de classes POJO de @Controller

Mapping ressource

```
@Controller
public class HomeController {
    @RequestMapping("/home")
    public String home(Model model) {
        model.addAttribute("message", "Allô le monde ?!");

        return "home";
    }
}
```

- « home » fait référence à une vue JSP « home.jsp »
- "message" pourra être lu depuis la vue JSP avec les EL \${ message }

# CONFIGURATION (VIEW)

## Choisir sa technologie

- Par exemple JSP/JSTL (on se base sur **JSTL**, on a besoin de la dépendance !)

## Paramétrer les vues dans dispatcher-context.xml avec un ViewResolver

- Nos vues JSP sont dans le répertoire "/WEB-INF/views/jsp/"
- Le nom des fichiers se terminent par ".jsp"
- Permettra de retourner "home" au lieu de "/WEB-INF/views/jsp/home.jsp" dans le contrôleur

```
<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/views/jsp/" />
  <property name="suffix" value=".jsp" />
</bean>
```



# CONFIGURATION (VIEW)

## Définition d'un @Bean

```
@Bean
public UrlBasedViewResolver viewResolver() {
    UrlBasedViewResolver viewResolver = new UrlBasedViewResolver();

    viewResolver.setViewClass(JstlView.class);
    viewResolver.setPrefix("/WEB-INF/views/jsp/");
    viewResolver.setSuffix(".jsp");

    return viewResolver;
}
```

# EXERCICE

Créer un nouveau projet

Configurer Spring MVC

- Dépendance **spring-webmvc**

Implémenter un @Controller **HomeController**

- Mapper une ressource /home
- Passer un attribut « nomUtilisateur » au modèle
- Afficher une JSP « home.jsp » qui affiche « Bonjour nomUtilisateur ! »



# CONTROLLER

Le contrôleur

# LE CONTROLLER

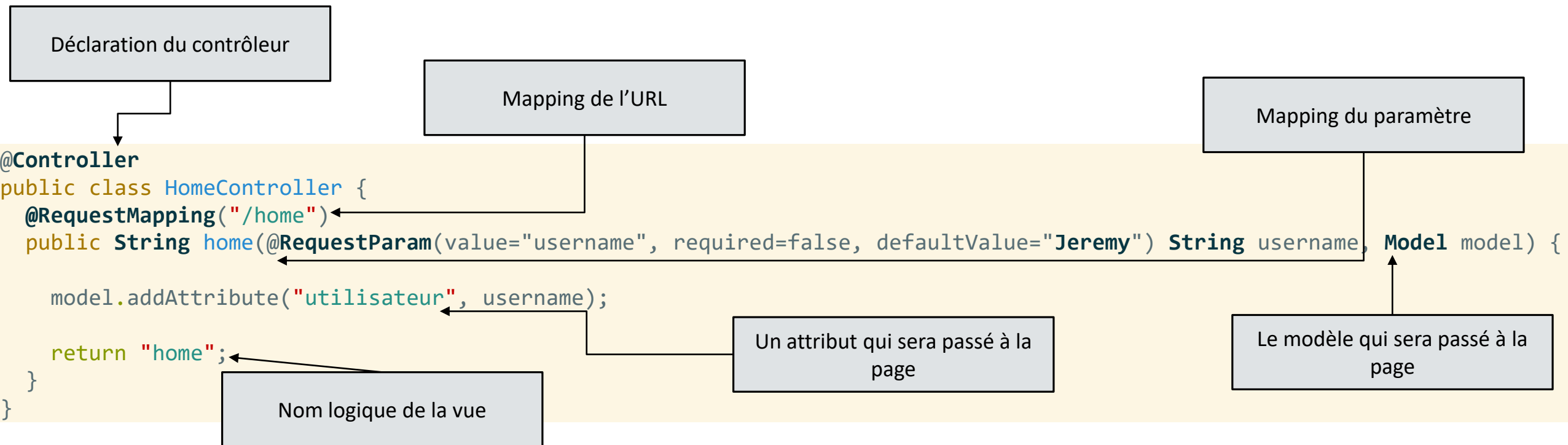
Le contrôleur est utilisé pour traiter une requête

Spring permet

- De mapper l'URL de la requête à une méthode d'un contrôleur
- De mapper les paramètres de la requête aux paramètres de la méthode
- De mapper des objets du contexte de la requête avec les paramètres de la méthode

# LE CONTROLLER

- `http://.../home` affichera « Bonjour JérémY ! »
- `http://.../home?username=Toto` affichera « Bonjour Toto ! »



Note : on peut rediriger en utilisant "redirect:url" (exemple : "redirect:/account/subscribe")

# LE CONTROLLER

La méthode mappée retourne une chaîne de caractères

- Utilisée pour retourner le nom de la vue
- Peut être utilisée pour rediriger l'utilisateur vers une autre adresse URL
  - Avec "redirect:/url"

```
@Controller
public class HomeController {
    @RequestMapping("/home")
    public String home(@RequestParam("username") String username, Model model) {
        return "redirect:/login";
    }
}
```

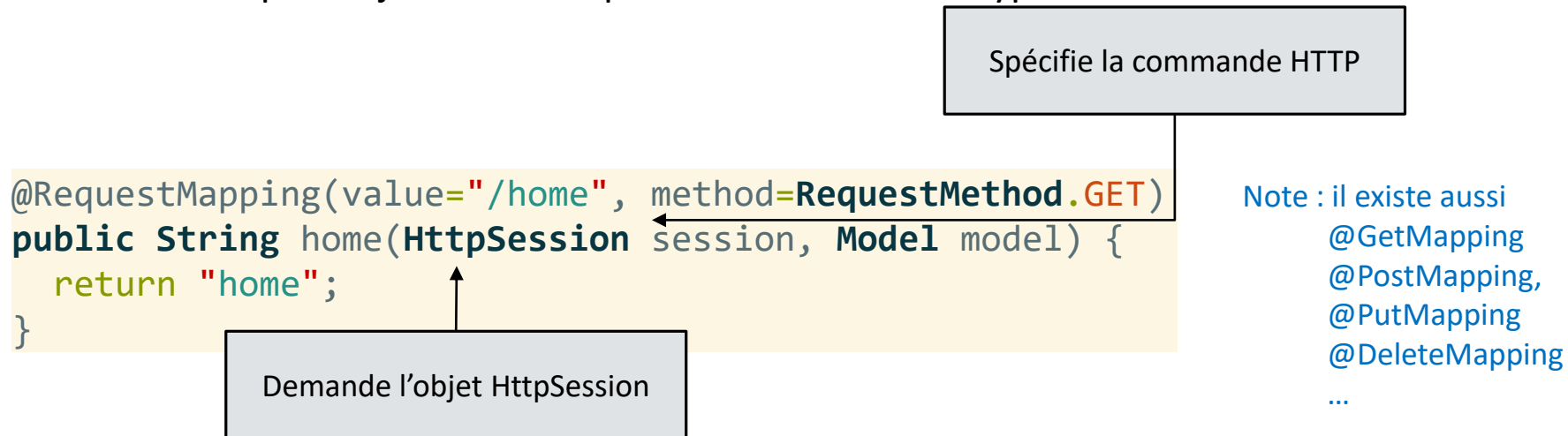
# LE CONTROLLER

Spécifier le type de commande HTTP (GET, POST, PUT, DELETE, ...)

- Par défaut, toutes les commandes HTTP sont mappées sur l'URL avec `@RequestMapping`

Demander l'objet `HttpSession` (qui nous permettra de manipuler la session)

- Principe d'injection de dépendance basé sur le type



# LE CONTROLLER

Variables dans la ressource et multiple mapping d'URL

- `http://.../home` affichera « Bonjour ! »
- `http://.../home/Toto` affichera « Bonjour Toto ! »

```
@GetMapping(value={ "/home", "/home/{username}" })  
public String home(@PathVariable(value="username", required=false) String username, Model model) {  
    model.addAttribute("utilisateur", username);  
  
    return "home";  
}
```



# LE CONTROLLER

Il est possible de mapper toutes les méthodes d'un contrôleur

- En annotant le contrôleur de `@RequestMapping`

```
@Controller
@RequestMapping("/produit")
public class ProduitController {
    @GetMapping("/liste")
    public String getProduits() {
        return "produits";
    }

    @GetMapping("/{nomProduit}")
    public String getProduit(@PathVariable(value="nomProduit") String produit) {
        return "produit";
    }
}
```

http://.../produit/liste → produits.jsp  
http://.../produit/gopro → produit.jsp  
http://.../produit/iphone → produit.jsp

# EXERCICE

Modifier la méthode « home » du contrôleur

- Doit attendre un paramètre « idProduit »
- Doit attendre une variable de chemin dans l'URL « username »
- Ne fonctionne qu'en méthode GET

Afficher sur la JSP la valeur de « idProduit » et de « username »

RAPPEL : pour utiliser plusieurs paramètres dans l'URL :

- `url?param1=valeur1&param2=valeur2`



# THYMELEAF

Configurer Thymeleaf

# THYMELEAF

Thymeleaf est un moteur de rendu pleinement compatible avec Spring

- Dépendances **thymeleaf-spring5** et **thymeleaf-layout-dialect**

## Configuration

- D'un **ViewResolver**
- D'un **TemplateEngine**
- D'un **TemplateResolver**

# THYMELEAF

## Coniguration du Bean du **TemplateResolver**

```
<bean id="templateResolver" class="org.thymeleaf.spring5.templateresolver.SpringResourceTemplateResolver">
  <property name="prefix" value="/WEB-INF/templates/" />
  <property name="suffix" value=".html" />
  <property name="templateMode" value="HTML" />
</bean>
```

# THYMELEAF

Bean du **TemplateEngine** à configurer

- Pour lequel on donnera la référence du bean **TemplateResolver**
- On lui ajoutera également le dialect **LayoutDialect**

```
<bean id="templateEngine" class="org.thymeleaf.spring5.SpringTemplateEngine">
  <property name="templateResolver" ref="templateResolver" />
  <property name="enableSpringELCompiler" value="true" />
  <property name="additionalDialects">
    <set>
      <bean class="nz.net.ultraq.thymeleaf.LayoutDialect"/>
    </set>
  </property>
</bean>
```

# THYMELEAF

## Bean du **ViewResolver** dans Spring

- Pour lequel on donnera la référence du bean **TemplateEngine**

```
<bean id="viewResolver" class="org.thymeleaf.spring5.view.ThymeleafViewResolver">  
  <property name="templateEngine" ref="templateEngine" />  
</bean>
```

# THYMELEAF

## Définition du @Bean **TemplateResolver**

```
@Bean
public SpringResourceTemplateResolver templateResolver() {
    SpringResourceTemplateResolver templateResolver = new SpringResourceTemplateResolver();

    templateResolver.setPrefix("/WEB-INF/templates/");
    templateResolver.setSuffix(".html");

    return templateResolver;
}
```



# THYMELEAF

## Définition du @Bean **TemplateEngine**

```
@Bean
public SpringTemplateEngine templateEngine(SpringResourceTemplateResolver templateResolver) {
    SpringTemplateEngine templateEngine = new SpringTemplateEngine();

    templateEngine.setTemplateResolver(templateResolver);
    templateEngine.setEnableSpringELCompiler(true);
    templateEngine.addDialect(new LayoutDialect());

    return templateEngine;
}
```

# THYMELEAF

## Définition du @Bean **ViewResolver**

```
@Bean
public ThymeleafViewResolver viewResolver(SpringTemplateEngine templateEngine) {
    ThymeleafViewResolver viewResolver = new ThymeleafViewResolver();

    viewResolver.setTemplateEngine(templateEngine);
    return viewResolver;
}
```

# EXERCICE

Implémenter Thymeleaf

Mettre en place le layout (la structure) du site

- Basé sur Bootstrap
- Titre
- Navigation (menu)
- Contenu dynamique



# DATA BINDING

Data Binding

# DATA BINDING

Des données issues d'un formulaire

- Il faut mapper les propriétés de la classe une à une
  - `setNom(request.getParameter("nom"))`
  - `setPrenom(request.getParameter("prenom"))`
  - ...

Le Data Binding va nous éviter tout ça !

Utilisation de l'annotation `@ModelAttribute("nom_model")`

# DATA BINDING

`@ModelAttribute("user")` associe les paramètres du formulaire à l'objet

- C'est Spring qui se chargera d'utiliser les setters de Utilisateur, à notre place

```
@PostMapping("/account/subscribe")
public String subscribe(@ModelAttribute("user") Utilisateur utilisateur, Model model) {
    System.out.println("Prénom : " + utilisateur.getPrenom() + " Nom : " + utilisateur.getNom());
    //...

    return "account/subscribe";
}
```

Récupérer l'objet bindé via  
l'attribut "user" de Model

Note : `@ModelAttribute` n'est pas obligatoire  
Spring cherchera un attribut du même nom que le paramètre  
Ou, le cas échéant, cherchera un attribut du même type

# DATA BINDING

```
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form" %>

<form:form method="post" action="subscribe" modelAttribute="user">
  <table>
    <tr>
      <td><form:label path="prenom">Prénom</form:label></td>
      <td><form:input path="prenom" /></td>
    </tr>

    <tr>
      <td><form:label path="nom">Nom</form:label></td>
      <td><form:input path="nom" /></td>
    </tr>

    <tr>
      <td colspan="2"><input type="submit" value="S'inscrire" /></td>
    </tr>
  </table>
</form:form>
```

Nom du @ModelAttribute

Nom des propriétés

# DATA BINDING

```
<form action="subscribe" method="post">
  <table>
    <tr>
      <td><label for="prenom">Prénom</label></td>
      <td><input id="prenom" name="prenom" type="text" value="{ user.prenom }" /></td>
    </tr>

    <tr>
      <td><label for="nom">Nom</label></td>
      <td><input id="nom" name="nom" type="text" /></td>
    </tr>

    <tr>
      <td colspan="2"><input type="submit" value="S'inscrire"></td>
    </tr>
  </table>
</form>
```

Objet du modèle et la valeur de sa propriété

Nom des propriétés



# DATA BINDING

L'attribut *ModelAttribute* du formulaire Spring nécessite un attribut par défaut

- Il faut donc le créer et l'ajouter au Model
  - Soit à l'affichage du formulaire

```
@GetMapping("/account/subscribe")
public String subscribe(Model model) {
    model.addAttribute("user", new Utilisateur());
    return "account/subscribe";
}
```

- Soit via une méthode annotée de *@ModelAttribute* (directement dans le contrôleur)

```
@ModelAttribute("user")
public Utilisateur initUtilisateur() {
    Utilisateur utilisateur = new Utilisateur();
    return utilisateur;
}
```

# EXERCICE

Créer une classe Utilisateur (nom, prénom, username, password)

Créer un contrôleur « AccountController »

- Ajouter le mapping « account/subscribe » GET
- Ajouter le mapping « account/subscribe » POST
  - Cette méthode affichera les informations saisies en console (System.out)
- Utiliser l'annotation @ModelAttribute pour initialiser l'attribut Utilisateur

Ajouter une page d'inscription JSP « subscribe.jsp »



# VALIDATION

Validation

# VALIDATION

## Le principe de la validation

- Vérifier si les champs obligatoires sont remplis
- Vérifier si une valeur est comprise entre x et y
- Ces validations sont à faire, une à une, dans la méthode POST

Spring MVC et l'API de validation vont nous éviter tout ça !

Utilisation de l'annotation `@Valid`

# VALIDATION

Ajouter @Valid devant @ModelAttribute

Active la validation api-validator  
(hibernate-validator)

```
@PostMapping("/account/subscribe")
public String subscribe(@Valid @ModelAttribute("user") Utilisateur utilisateur, BindingResult result, Model model) {
    if (result.hasErrors()) {
        System.out.println("L'utilisateur n'a pas été validé ...");
        return "subscribe";
    }

    System.out.println("Prénom : " + utilisateur.getPrenom() + " Nom : " + utilisateur.getNom());
    return "account/subscribe";
}
```

Permet de connaître les erreurs lors du Bind,  
si erreur il y a

Si utilisation de plusieurs @ModelAttribute

- Il faut placer un BindingResult juste après un @ModelAttribute
  - Celui qui suit @ModelAttribute lui correspond

# VALIDATION

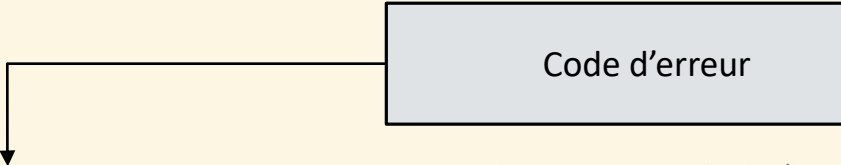
2 options sont possibles pour la validation

- Utiliser une classe qui implémente l'interface "Validator"
- Utiliser Hibernate-Validation

# VALIDATION — VALIDATOR

Implémenter une classe qui implémente Validator (SpringFramework)

```
public class UtilisateurSubscribeValidator implements Validator {  
    @Override public boolean supports(Class<?> cls) {  
        return Utilisateur.class.equals(cls);  
    }  
  
    @Override public void validate(Object obj, Errors e) {  
        ValidationUtils.rejectIfEmptyOrWhitespace(e, "prenom", "prenom.empty", "Le prénom doit être saisi");  
        ValidationUtils.rejectIfEmptyOrWhitespace(e, "nom", "nom.empty", "Le nom doit être saisi");  
    }  
}
```



Code d'erreur

# VALIDATION — VALIDATOR

## La méthode `@RequestMapping` nécessite un Validator

- Soit en première instruction de la méthode (dans ce cas, `@Valid` n'est plus nécessaire)

```
@PostMapping("/account/subscribe")
public String subscribe(@Valid @ModelAttribute("user") Utilisateur utilisateur, BindingResult result, Model model) {
    new UtilisateurSubscribeValidator().validate(utilisateur, result);
    //...
}
```

- Soit via une méthode annotée de `@InitBinder` (directement dans le contrôleur)

```
@InitBinder
protected void initBinder(WebDataBinder binder) {
    binder.addValidators(new UtilisateurSubscribeValidator());
}
```



# VALIDATION — HIBERNATE-VALIDATOR

Il suffit d'annoter les propriétés de la classe

```
public class Utilisateur {  
    @NotEmpty(message = "Le nom est obligatoire")  
    private String nom;  
  
    @NotEmpty(message = "Le prénom est obligatoire")  
    private String prenom;  
}
```

# VALIDATION

Pour afficher les messages d'erreur dans la page JSP

- `<form:errors path="nom_propriete" />`
  - *Doit être obligatoirement contenu dans un `<form:form />`*

```
<form:form method="post" action="subscribe" modelAttribute="user">
  <table>
    <tr>
      <td><label for="prenom">Prénom</label></td>
      <td><input id="prenom" name="prenom" type="text" value="${ user.prenom }" /></td>
      <td><form:errors path="prenom" /></td>
    </tr>
  </table>
</form:form>
```

# EXERCICE

## Mettre en place la validation pour l'inscription

- Le nom, le prénom, le nom d'utilisateur et le mot de passe sont obligatoires
- Le mot de passe et le mot de passe de vérification doivent correspondre
- Combinez la validation par Hibernate-Validator et la validation par une classe Validator
  - Utilisez `e.rejectValue()` dans ce Validator pour déclencher la non-validation



**I18N**

Internationalisation i18n

# INTERNATIONALISATION

Utilisation de la notion de « Locale », basé sur Pays\_langue

- FR\_fr, US\_en, ...

Utilisation de constantes dans le code source

Définition des constantes dans un fichier de ressources

- Pair = valeur
- Un fichier par langue
- Un fichier par défaut
- Le nom des fichiers sera formé de la même façon
  - messages\_fr.properties
  - messages\_en.properties
  - messages\_de.properties
  - ...

# INTERNATIONALISATION

Ces fichiers « properties », placés dans les ressources du projet

- Contiendront un code et une valeur

```
title.home = Welcome to our amazing page!  
page.home.welcome = Welcome, {0}!
```

```
title.home = Bienvenue sur notre superbe page !  
page.home.welcome = Bienvenue {0} !
```

# INTERNATIONALISATION

Il sera possible de les rechercher dans les JSP

- Utiliser la taglib, préfixée « spring »
- Utiliser la balise « spring:message »

```
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring" %>

<p>
  <spring:message code="page.home.welcome" arguments="${ utilisateur }" />
</p>
```

# INTERNATIONALISATION

Il faut demander à Spring de gérer l'internationalisation

- En déclarant un bean à cet effet

```
<bean id="messageSource" class="org.springframework.context.support.ReloadableResourceBundleMessageSource">  
  <property name="basename" value="classpath:messages" />  
  <property name="defaultEncoding" value="UTF-8" />  
</bean>
```



# INTERNATIONALISATION

## Déclaration d'un @Bean

```
@Bean
public ReloadableResourceBundleMessageSource messageSource() {
    ReloadableResourceBundleMessageSource messageSource = new ReloadableResourceBundleMessageSource();

    messageSource.setBasename("classpath:messages");
    return messageSource;
}
```

# INTERNATIONALISATION

Par défaut, la langue chargée sera la langue du navigateur

Pour modifier ce comportement et pouvoir changer à la volée la langue

- Utiliser LocaleChangeInterceptor et CookieLocaleResolver
  - Dans la configuration de Spring
  - Ci-dessous, il sera possible de modifier la langue en précisant le paramètre "lang" dans l'URL
    - /home?lang=fr, ou /home?lang=en

```
<mvc:interceptors>
  <mvc:interceptor>
    <mvc:mapping path="/**" />
    <bean id="localeChangeInterceptor" class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
      <property name="paramName" value="lang" />
    </bean>
  </mvc:interceptor>
</mvc:interceptors>

<bean id="localeResolver" class="org.springframework.web.servlet.i18n.CookieLocaleResolver">
  <property name="cookieName" value="lang" />
  <property name="defaultLocale" value="en" />
</bean>
```

# INTERNATIONALISATION

## Déclaration d'un @Bean

```
@Bean
public CookieLocaleResolver localeResolver() {
    CookieLocaleResolver localeResolver = new CookieLocaleResolver();

    localeResolver.setCookieName("lang");
    localeResolver.setDefaultLocale(Locale.FRANCE);
    return localeResolver;
}
```

# INTERNATIONALISATION

## Déclaration d'un intercepteur d'URL

```
@Override
public void addInterceptors(InterceptorRegistry registry) {
    LocaleChangeInterceptor localeChangeInterceptor = new LocaleChangeInterceptor();

    localeChangeInterceptor.setParamName("lang");
    registry.addInterceptor(localeChangeInterceptor);
}
```

# EXERCICE

Traduire la page d'accueil et la page d'inscription en anglais

Implémenter la possibilité de changer de langue pendant la navigation