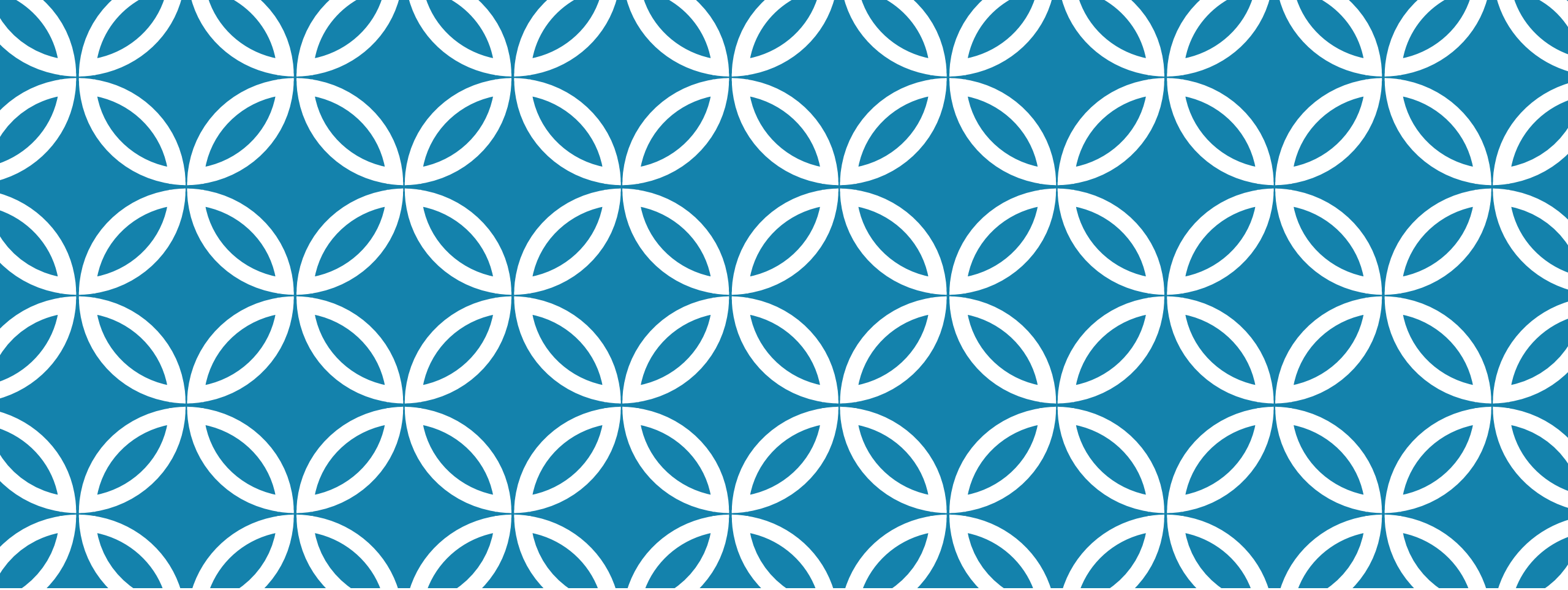
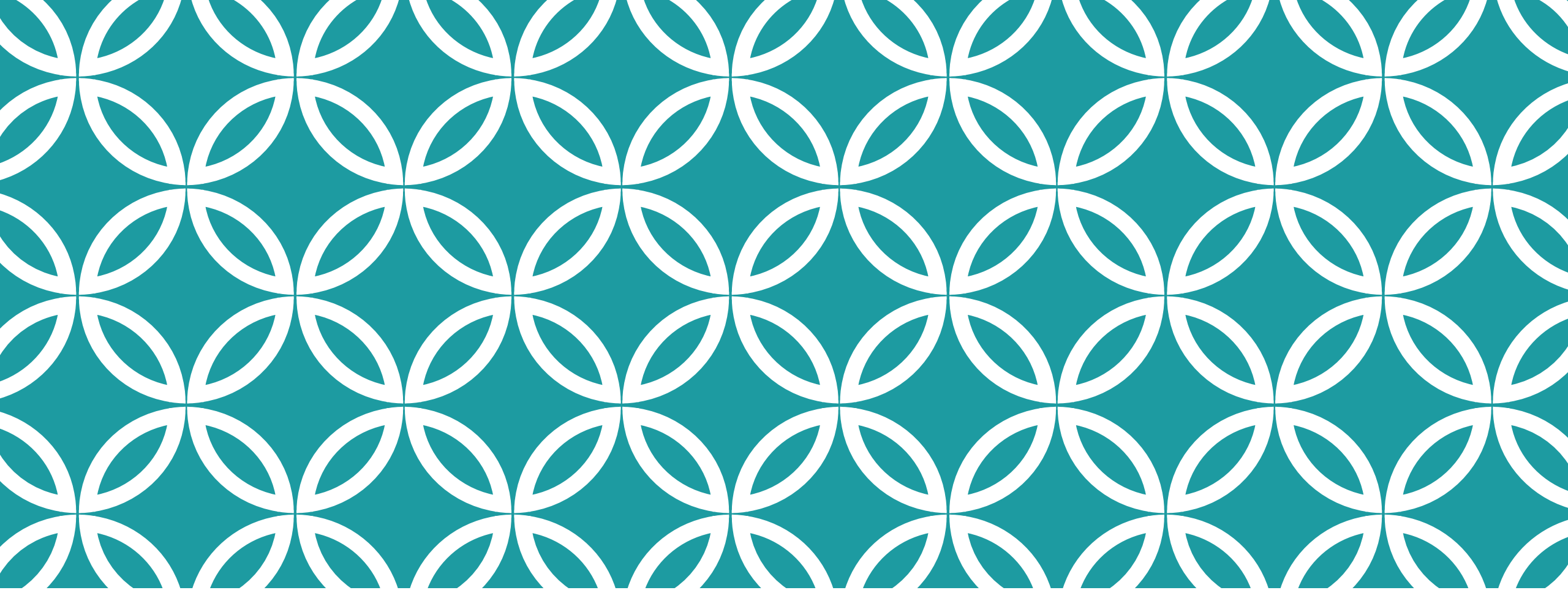


SPRING

Jérémy PERROUAULT

02/02/2022
Version 3



SPRING WEB MVC

Introduction à
Spring Web MVC

PRÉSENTATION DE SPRING MVC

Une Servlet principale : **DispatcherServlet**

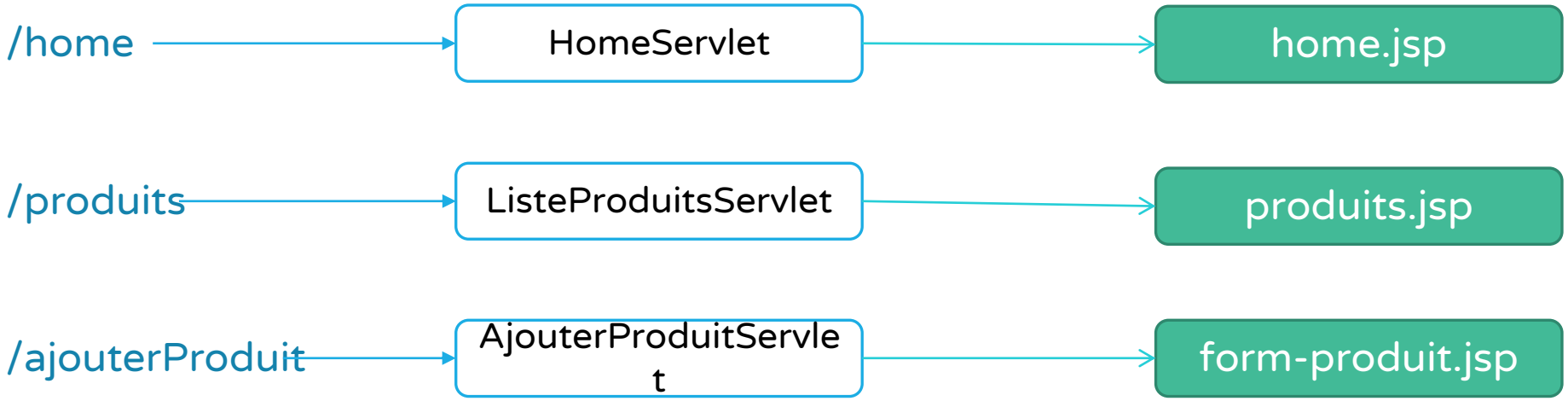
- Délègue les requêtes à des contrôleurs (classes annotées @**Controller**)
  - Selon le point d'accès (la ressource URL)

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)

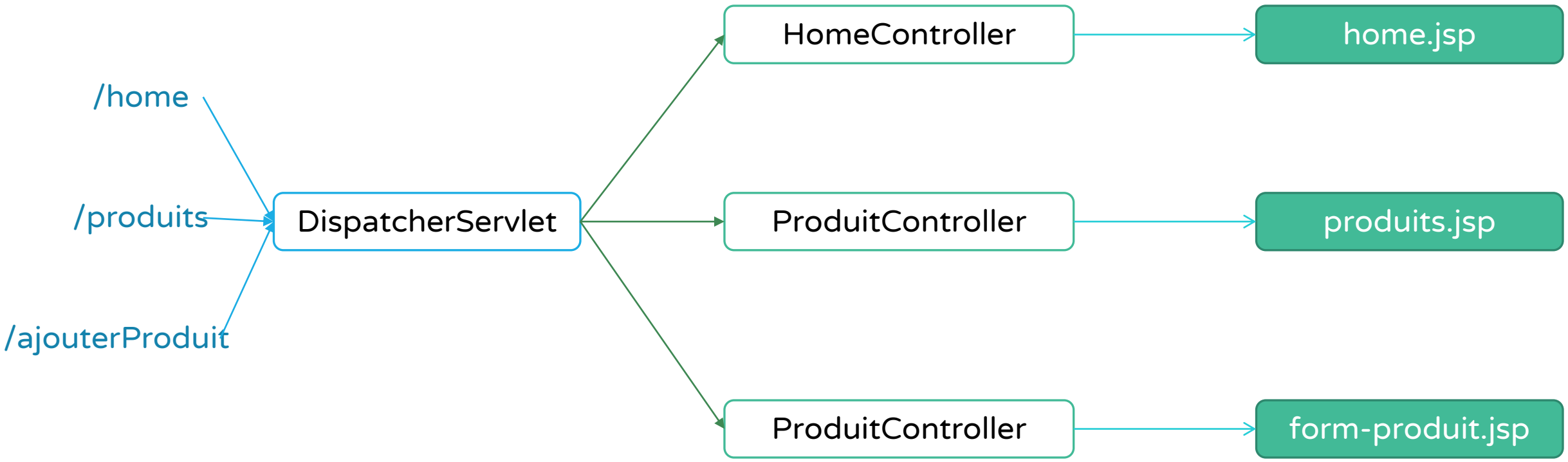
PRÉSENTATION DE SPRING MVC

JEE Servlets classiques



PRÉSENTATION DE SPRING MVC

Avec Spring MVC



PRÉSENTATION DE SPRING MVC

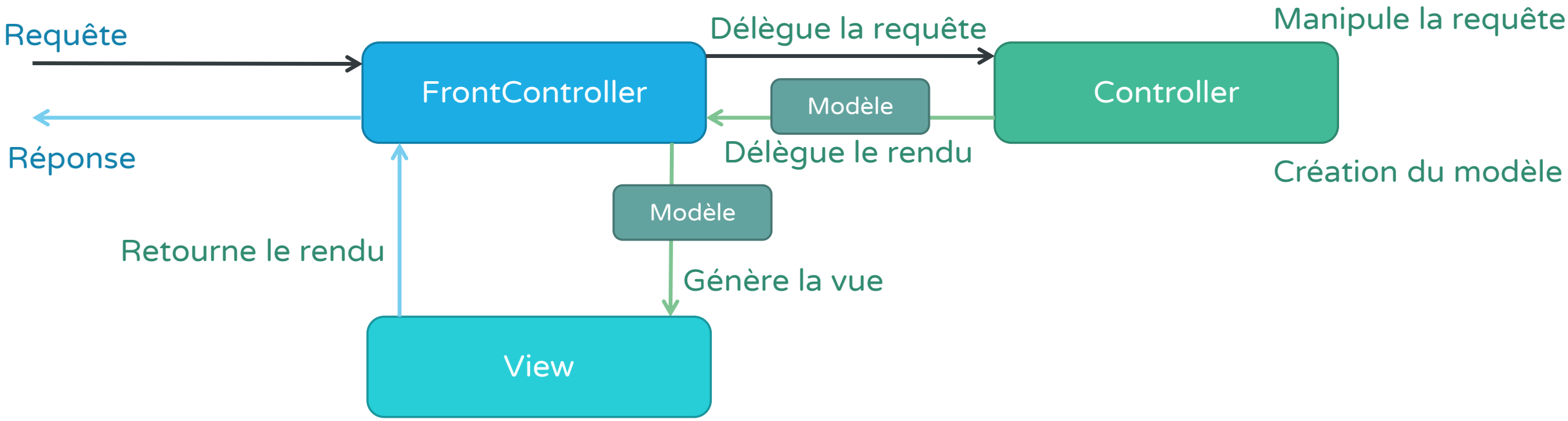
La Servlet **DispatcherServlet** est mappée sur un pattern

- Par exemple « / » (toutes les URL)
- On l'appelle « FrontController »

Il n'y a plus nécessité des Servlets JEE

- Mais si elles existent, leur mapping prend le pas sur le mapping des @Controller !
- Tout est géré dans le contexte de Spring

TRAITEMENT D'UNE REQUÊTE



CONFIGURATION

Mise en place de la configuration

CONFIGURATION

Pour une utilisation dans un cadre Application Web MVC

- Dépendance **spring-webmvc**
- Déclaration de la Servlet **DispatcherServlet** dans web.xml

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>
```

CONFIGURATION (FRONT CONTROLLER)

```
<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.AppConfig</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 configurer le profile, on ajoute un paramètre dans le **DispatcherServlet**

```
<init-param>
  <param-name>spring.profiles.active</param-name>
  <param-value>dev</param-value>
</init-param>
```

CONFIGURATION (FRONT CONTROLLER)

Dans le fichier de configuration de **DispatcherServlet**

- ➔ dispatcher-context.xml
- ➔ WebConfig.java

- 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

```
@Configuration
@EnableWebMvc
public class WebConfig {

}
```

EXERCICE

Créer un nouveau projet « eshop-web » (Maven)

Implémenter et configurer Spring MVC

- Dépendance **spring-webmvc**

Déployer le projet sur le serveur

- Il ne doit pas y avoir d'erreur au démarrage du serveur !

CONFIGURATION (RESOURCES)

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)

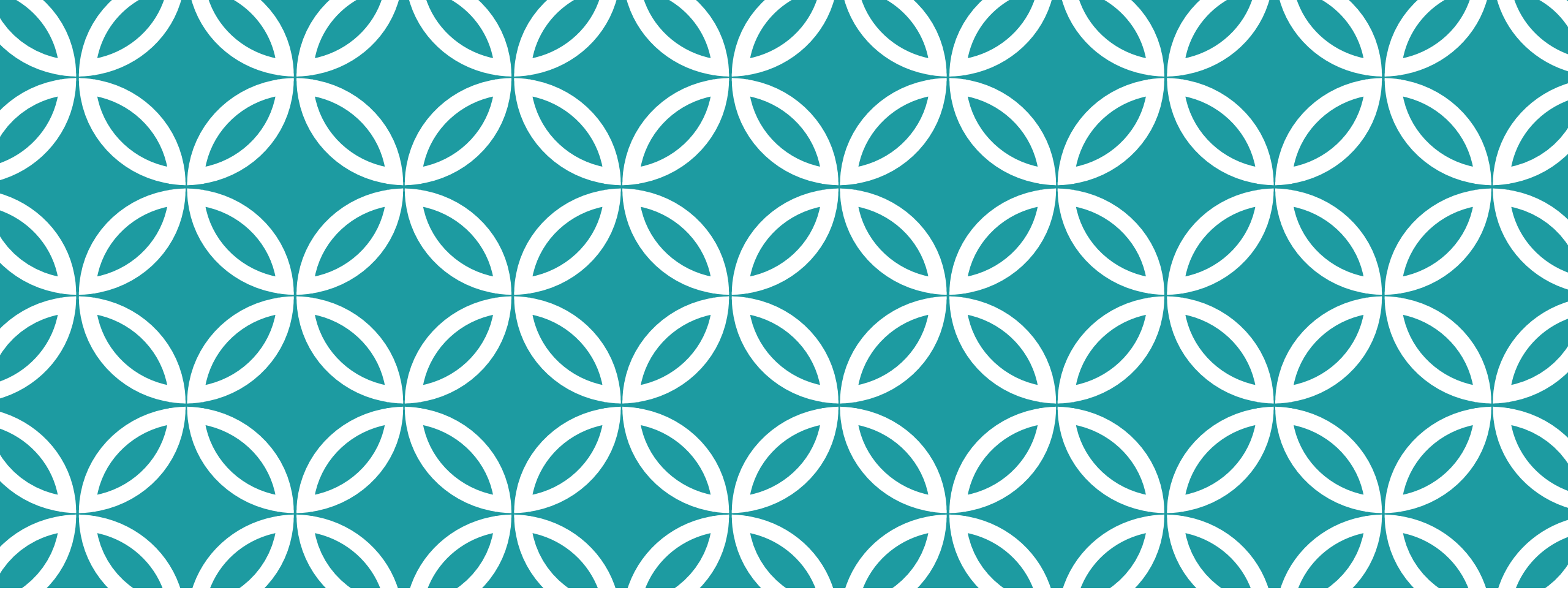
```
<mvc:resources mapping="/assets/**" location="/WEB-INF/assets/" />
```

CONFIGURATION (RESOURCES)

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("/assets/**").addResourceLocations("/WEB-INF/assets/");
  }
}
```



CONTROLLER

Le contrôleur

LE CONTROLLER

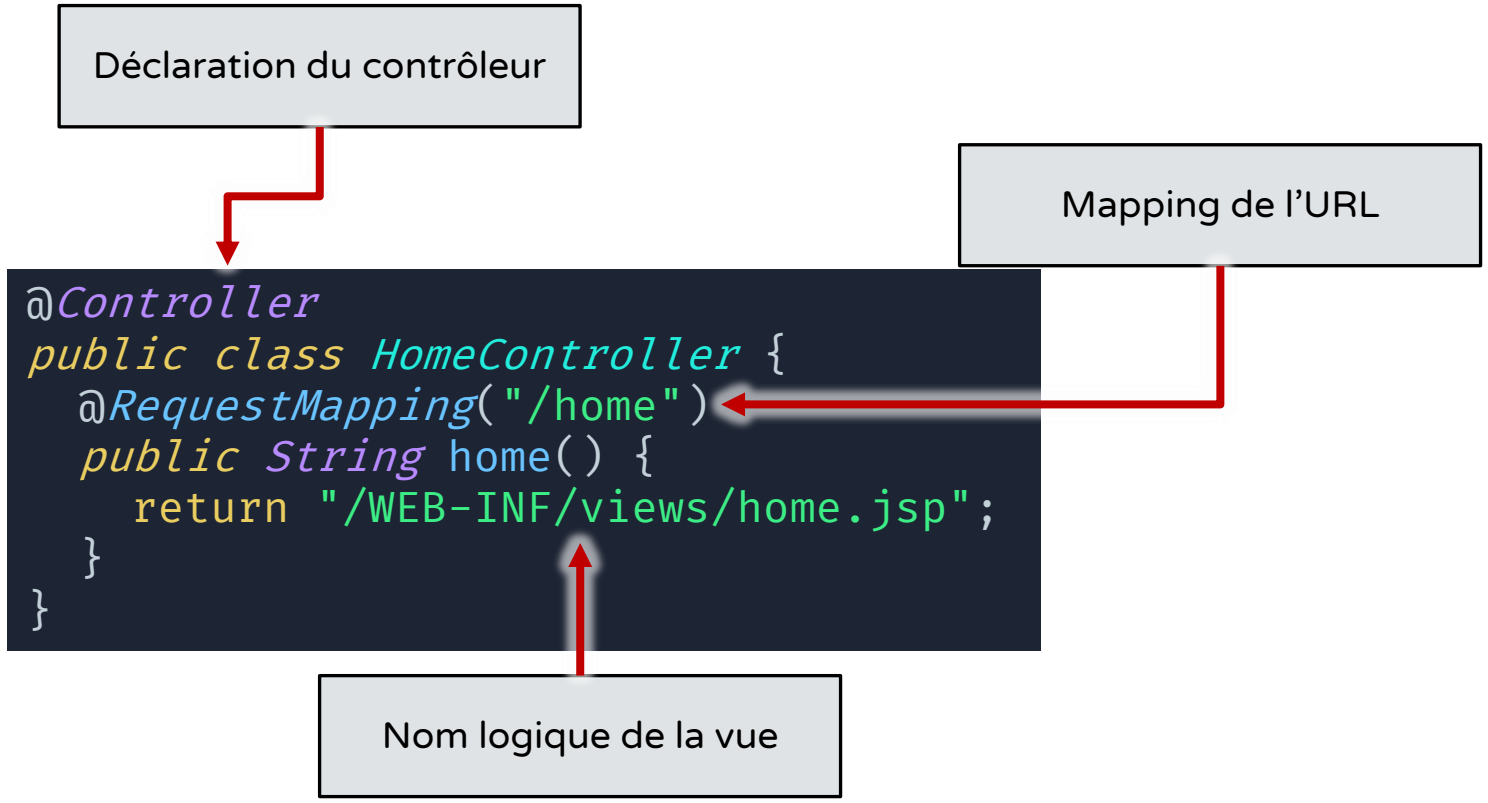
Le contrôleur est utilisé pour traiter une requête

- Classe POJO annotée de @**Controller**

Le mécanisme est le suivant :

- Mapping d'une URL sur une méthode d'un contrôleur
- Injection possible
  - Des paramètres de la requête aux arguments de la méthode
  - Des objets du contexte de la requête aux arguments de la méthode
  - De tout *bean* géré par Spring
- La méthode retourne une chaine de caractères
  - Il s'agit du nom logique de la vue que Spring doit générer
  - Spring va utiliser le *bean* **ViewResolver** déclaré dans le fichier de configuration

LE CONTROLLER



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

EXERCICE

Créer un contrôleur **HomeController**

- Méthode mappée sur « /home »
  - Affiche une JSP « home.jsp »
    - « Allô le monde ?! »

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/"
- Le nom des fichiers se terminent par ".jsp"
- Permettra de retourner "home" au lieu de "/WEB-INF/views/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/" />
  <property name="suffix" value=".jsp" />
</bean>
```

CONFIGURATION (VIEW)

En configuration Java, définition d'un @**Bean**

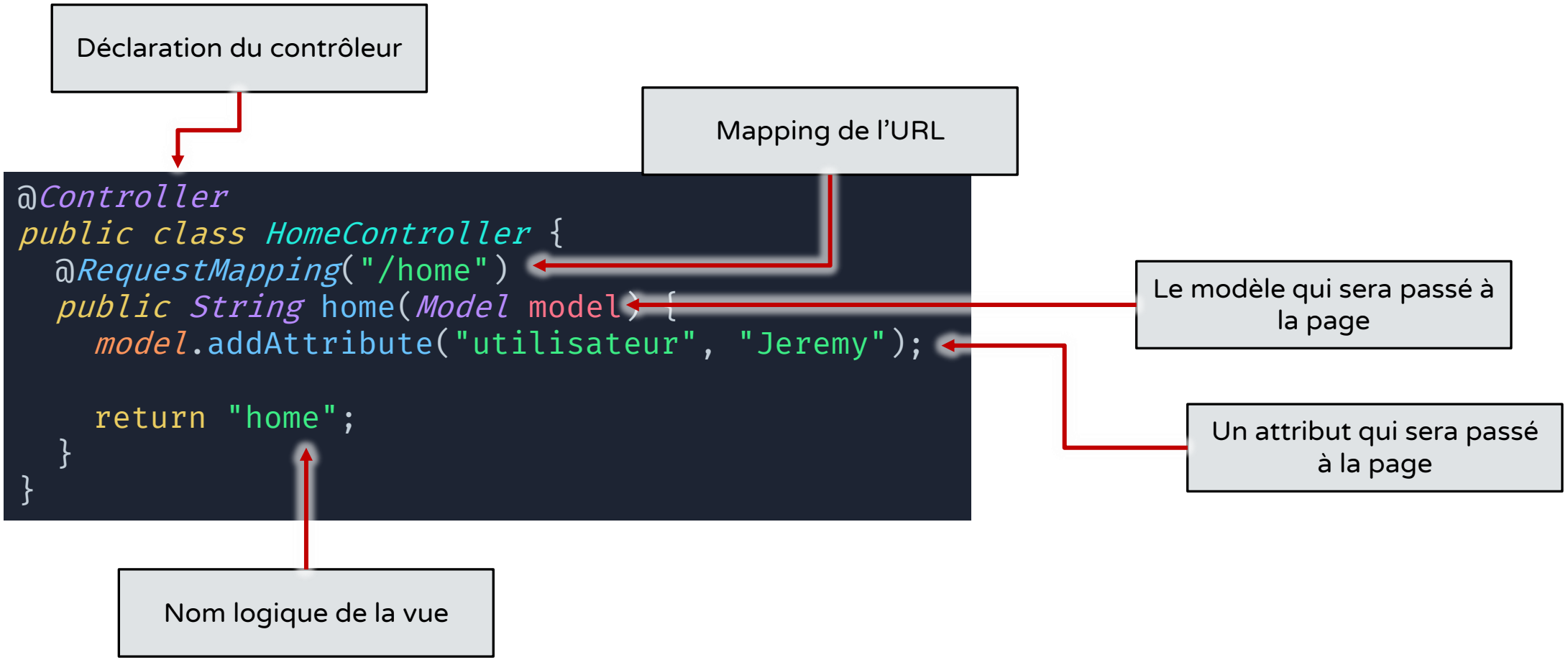
```java
@Bean
public ViewResolver viewResolver() {
  UrlBasedViewResolver viewResolver = new UrlBasedViewResolver();

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

  return viewResolver;
}
```

EXERCICE

Mettre en place la configuration du **ViewResolver**

- Modifier le contrôleur en conséquence

LE CONTROLLER




```java
@Controller
public class HomeController {
  @RequestMapping("/home")
  public String home(Model model) {
    model.addAttribute("utilisateur", "Jeremy");

    return "home";
  }
}
```

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 ProduitController {
  @RequestMapping("/ajouterProduit")
  public String addProduit() {
    return "redirect:/produits";
  }
}
```

LE CONTROLLER

Il est possible de 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

```
@RequestMapping(value="/home", method=RequestMethod.GET)
public String home(HttpSession session, Model model) {
  return "home";
}
```

Demande l'objet **HttpSession**

Note : il existe aussi
@GetMapping
@PostMapping
@PutMapping
@DeleteMapping
…

Spécifie la commande HTTP

EXERCICE

Créer un contrôleur **HomeController**

- Méthode mappée sur « /home »
  - N'accepte que du GET
  - Affiche une JSP « home.jsp »
    - « Allô le monde ?! »

LE CONTROLLER – PARAMÈTRES

Avec les Servlets

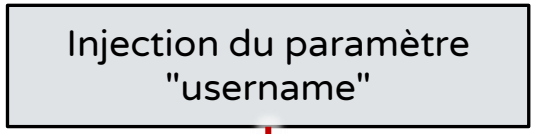
- Utilisation de l'objet **HttpServletRequest** pour récupérer un paramètre (getParameter)

Avec Spring

- Possible d'utiliser **HttpServletRequest**…
- Mais préférable d'utiliser l'injection, avec l'annotation @**RequestParam** !
  - Convertit automatiquement la valeur selon le type de l'argument

LE CONTROLLER – PARAMÈTRES

http://localhost:8080/mon-projet/home?username=Albert




```java
@Controller
public class HomeController {
  @RequestMapping("/home")
  public String home(@RequestParam(value="username", required=false, defaultValue="Jeremy") String name, Model model) {
    model.addAttribute("utilisateur", name);
    return "home";
  }
}
```

NOTE : Si le nom du paramètre ("username") est identique au nom de l'argument, Spring fait le lien tout seul

```java
@Controller
public class HomeController {
  @RequestMapping("/home")
  public String home(@RequestParam String username, Model model) {
    model.addAttribute("utilisateur", username);
    return "home";
  }
}
```

LE CONTROLLER – PARAMÈTRES

Spring est capable de convertir le paramètre selon le type de l'argument

```
@Controller
public class ProduitController {
  @RequestMapping("/ajouterProduit")
  public String addProduit(@RequestParam String nom, @RequestParam float prix) {
    //...
  }
}
```

EXERCICE

Modifier **HomeController**

- La méthode attend un paramètre optionnel "username"
  - N'accepte que du GET
  - Affiche une JSP « home.jsp »
    - « Bonjour nomUtilisateur ! »

Créer un contrôleur **ProduitController**

- Méthode mappée sur « /produit » qui attend un paramètre obligatoire "id" (entier)
  - N'accepte que du GET
  - Rechercher le produit qui a l'ID
  - Affiche une JSP « produit.jsp »
    - « Voici le libellé du produit : libelle »

LE CONTROLLER – VARIABLES DE CHEMIN

A l'instar des paramètres

- Les variables de chemin sont des informations envoyées par le client
- Sous une forme plus technique, directement incluse dans le chemin de l'URL

Exemples avec paramètres

- http://localhost:8080/mon-projet/produit?id=42
- http://localhost:8080/mon-projet/produit?id=42&nom=gopro-hero-6

Exemples avec variables de chemin

- http://localhost:8080/mon-projet/produit/42
- http://localhost:8080/mon-projet/produit/42-gopro-hero-6

LE CONTROLLER – VARIABLES DE CHEMIN

Les variables de chemin est les paramètres peuvent s'utiliser en même temps

- http://localhost:8080/mon-projet/produit/42-gopro-hero-6?couleur=noire

LE CONTROLLER – VARIABLES DE CHEMIN

Comme pour les paramètres et leur annotation @**RequestParam**

- Les variables de chemin ont leur annotation @**PathVariable** !

```
@GetMapping({ "/produit", "/produit/{produitId}" })
public String home(@PathVariable(value="produitId", required=false) int idProduit, Model model) {
  model.addAttribute("id", idProduit);
  return "produit";
}
```

- Et si le nom de la variable est identique au nom de l'argument …

```
@GetMapping({ "/produit", "/produit/{idProduit}" })
public String home(@PathVariable int idProduit, Model model) {
  model.addAttribute("id", idProduit);
  return "produit";
}
```

EXERCICE

Modifier **ProduitController**

- Méthode mappée sur « /produit » qui attend une variable obligatoire "id" (entier)
  - N'accepte que du GET
  - Rechercher le produit qui a l'ID
  - Affiche une JSP « produit.jsp »
    - « Voici les informations du produit : id, libelle et prix »

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("/{idProduit}")
  public String getProduit(@PathVariable String idProduit) {
    return "produit";
  }
}
```

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

LE CONTROLLER – STATUS HTTP

Il est possible de modifier le status HTTP de réponse

- En utilisant l'annotation @**ResponseStatus** sur la méthode

EXERCICE

Commencer le CRUD de « produit » dans un même contrôleur **ProduitController**

- Lister les produits
  - GET
- Ajouter un produit
  - GET / POST
- Supprimer un produit
  - GET

LE CONTROLLER – GESTION DES EXCEPTIONS

Un traitement peut lever une exception

- On peut choisir de la traiter dans le code avec les blocs *try .. catch* …
- Ou bien d'afficher un message d'erreur HTTP à l'utilisateur (status erreur)

- Créer une **RuntimeExeption** annotée de @**ResponseStatus**
- Lever cette **RuntimeException** si nécessaire
  - Le status HTTP sera modifié et l'erreur envoyée à l'utilisateur

LE CONTROLLER – GESTION DES EXCEPTIONS

Un traitement peut lever une exception

- On peut choisir de la traiter dans le code avec les blocs *try .. catch* …
- Ou bien dédier un contrôleur spécifique à la gestion des exceptions
  - Annoté de @**ControllerAdvice**
  - Les méthodes d'interception seront annotées de @**ExceptionHandler** et éventuellement de @**ResponseStatus**

```
@ControllerAdvice
public class ExceptionHandlerController {
  @ExceptionHandler(Exception.class)
  public String handleException(Exception ex) {
    return "pageErreur";
  }
}
```

LE CONTROLLER – GESTION DES EXCEPTIONS

On peut aussi gérer les ressources non trouvées (non mappées)

- Ajouter dans web.xml le paramètre *throwExceptionIfNoHandlerFound* avec la valeur à *true*
- Avoir une méthode qui intercepte l'exception **NoHandlerFoundException**

```
<init-param>
  <param-name>throwExceptionIfNoHandlerFound</param-name>
  <param-value>true</param-value>
</init-param>
```

```
@ControllerAdvice
public class ExceptionHandlerController {
  @ExceptionHandler(NoHandlerFoundException.class)
  @ResponseStatus(HttpStatus.NOT_FOUND)
  public String noHandlerFound() {
    return "pageErreur404";
  }
}
```

EXERCICE

Mettre en place la gestion des exceptions

- Page 404 personnalisée
- Page d'erreur générale