
# Modularité et réutilisation

**Chouki Tibermacine**

**Chouki.Tibermacine@umontpellier.fr**

# Plan de cette partie du cours

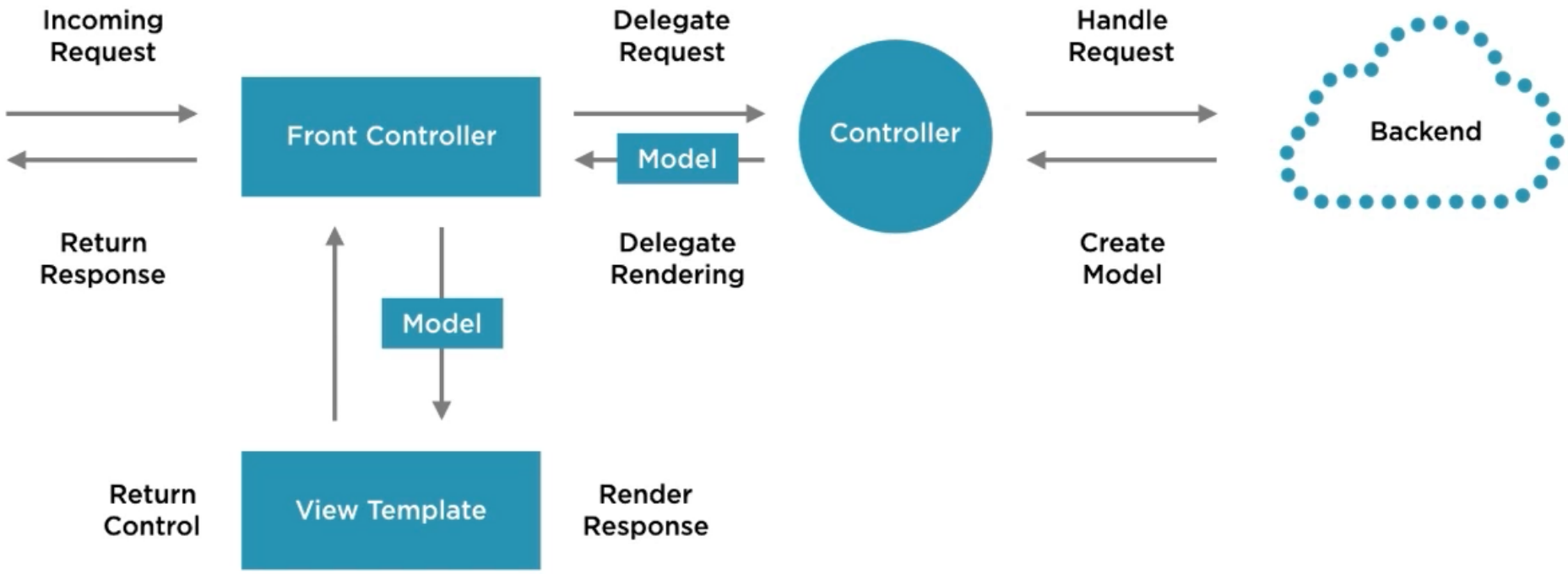
1. Système de build Maven
2. Introduction à la prog Web avec Java
3. Modules distribués Java (JEE)
4. Modularité à grain fin avec Spring DI
5. Applications Web Modulaires avec Spring MVC et Spring Boot
6. Système de modules à gros grain Java 9+

# Plan du cours

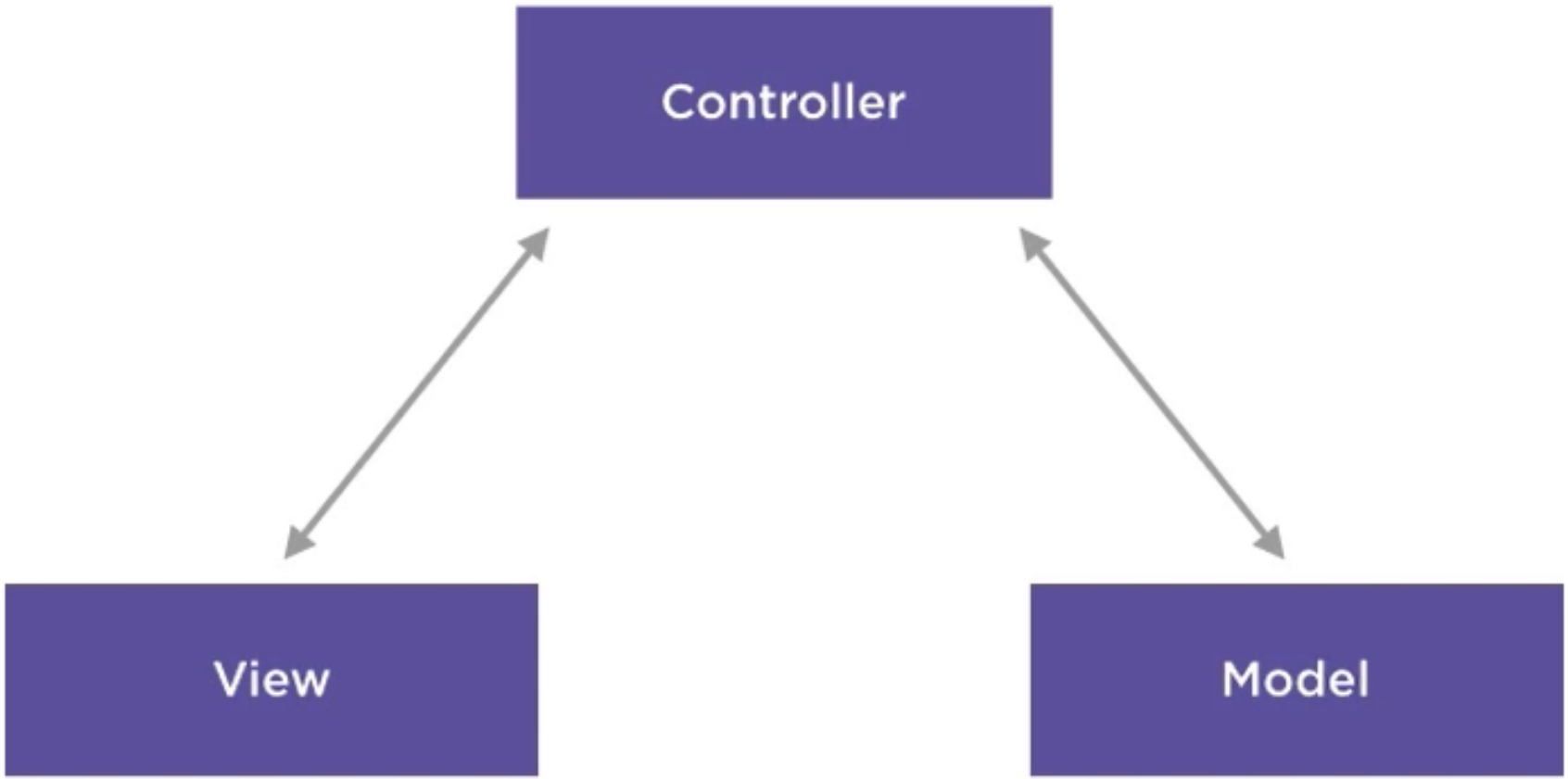
# Pourquoi Spring MVC ?

- Gérer la complexité de grandes applications, en séparant les rôles et en les répartissant sur différents composants (gérant le mapping des requêtes, la résolution des vues, la validation des modèles, ...)
- Programmation simplifiée par configuration (instanciation automatique, recherche d'instances, ... – cours Spring DI)
- Besoin de développer des applications avec une API REST ou des applications SPA (*Single Page*)
- Besoin de concevoir des applications indépendamment d'une technologie de templates (HTML, JSP, ...) pour le front-end ou de persistance des données pour le back-end (JDBC, JPA, Hibernate, ...)

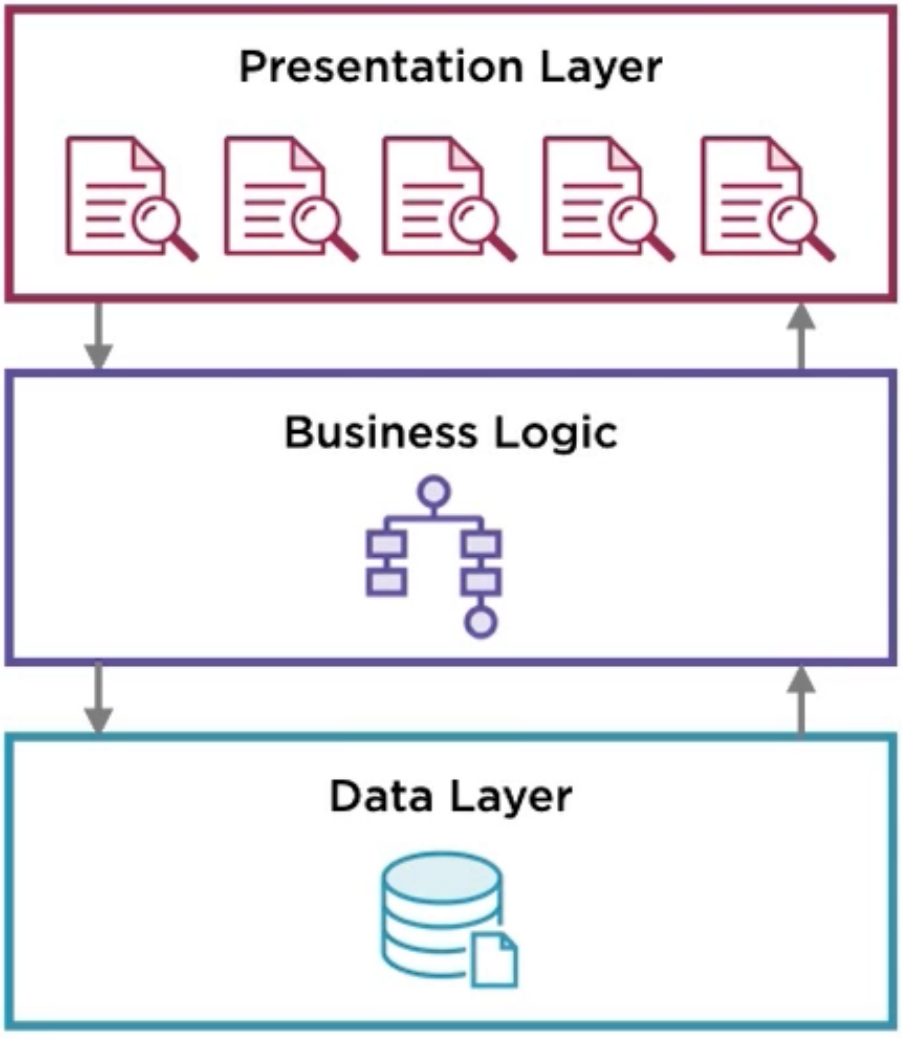
# Rappel sur le cycle de vie d'une requête/réponse HTTP



# Les contrôleurs au cœur des interactions



# Les contrôleurs au cœur des interactions

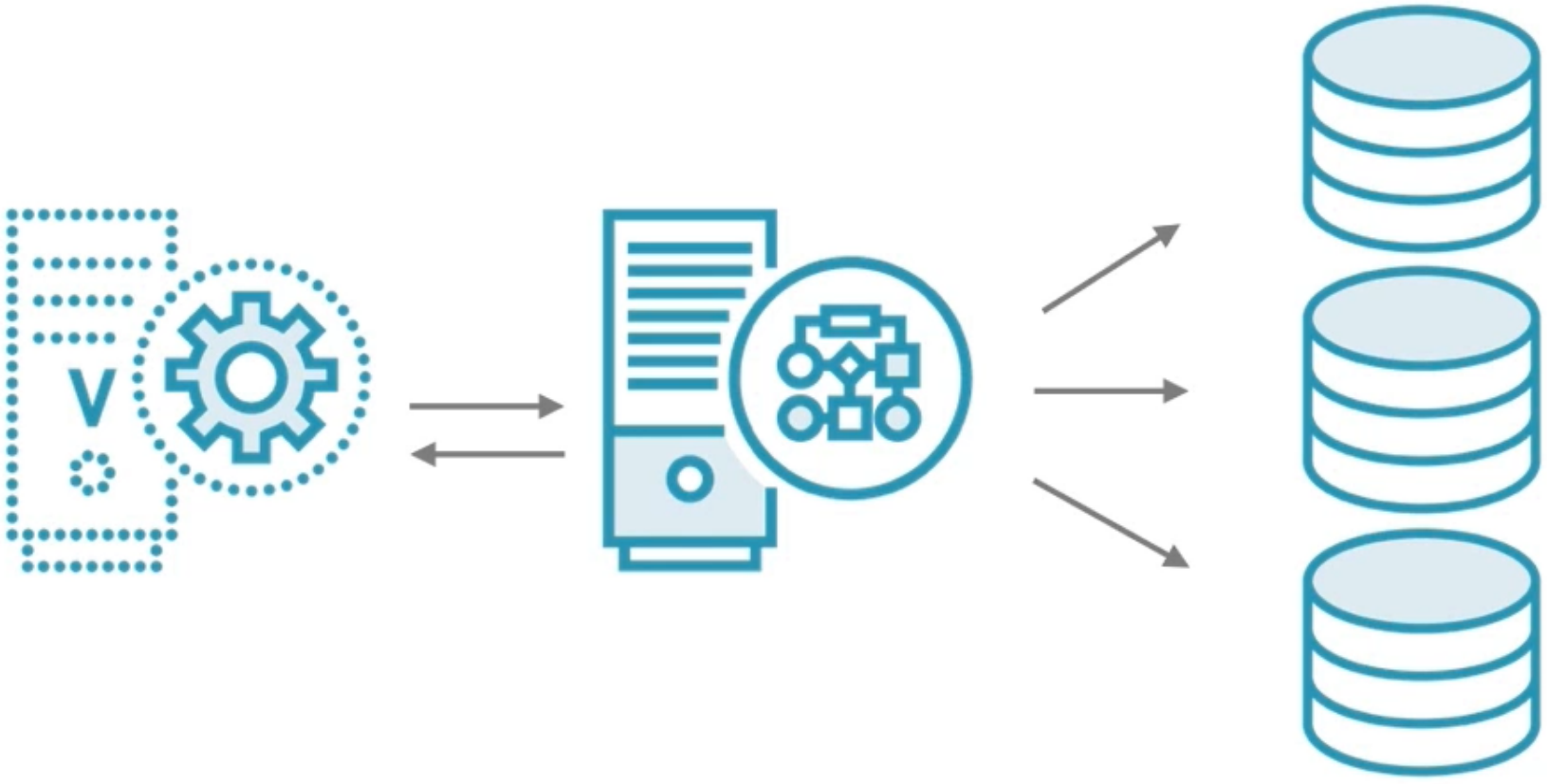



View (JSPs)

Controller (@Controller)

Data Model / Database
(Model Object)

# Des services derrière les contrôleurs



Contrôleurs Services Repositories

# Composant “*Contrôleur*” dans Spring MVC

- Gère les requêtes/réponses
- Pas de logique métier implémentée dedans
- Coordonne les services
- Classe annotée par `@Controller`
- Gère les exceptions et le routage vers les vues

# Composant “*Service*” dans Spring MVC

- Classe annotée `@Service`
- Décrit les verbes/actions de l’application
- Là où réside la logique métier
- Garantit un état cohérent des objets métier
- Là où démarrent/s’arrêtent les transactions

# Composant “*Repository*” dans Spring MVC

- Classe annotée `@Repository`
- Décrit les noms (pas les verbes) ou les données de l’application
- Là où se déroulent les interactions avec les bases de données
- Mapping 1 à 1 avec les objets métier

# Plan du cours

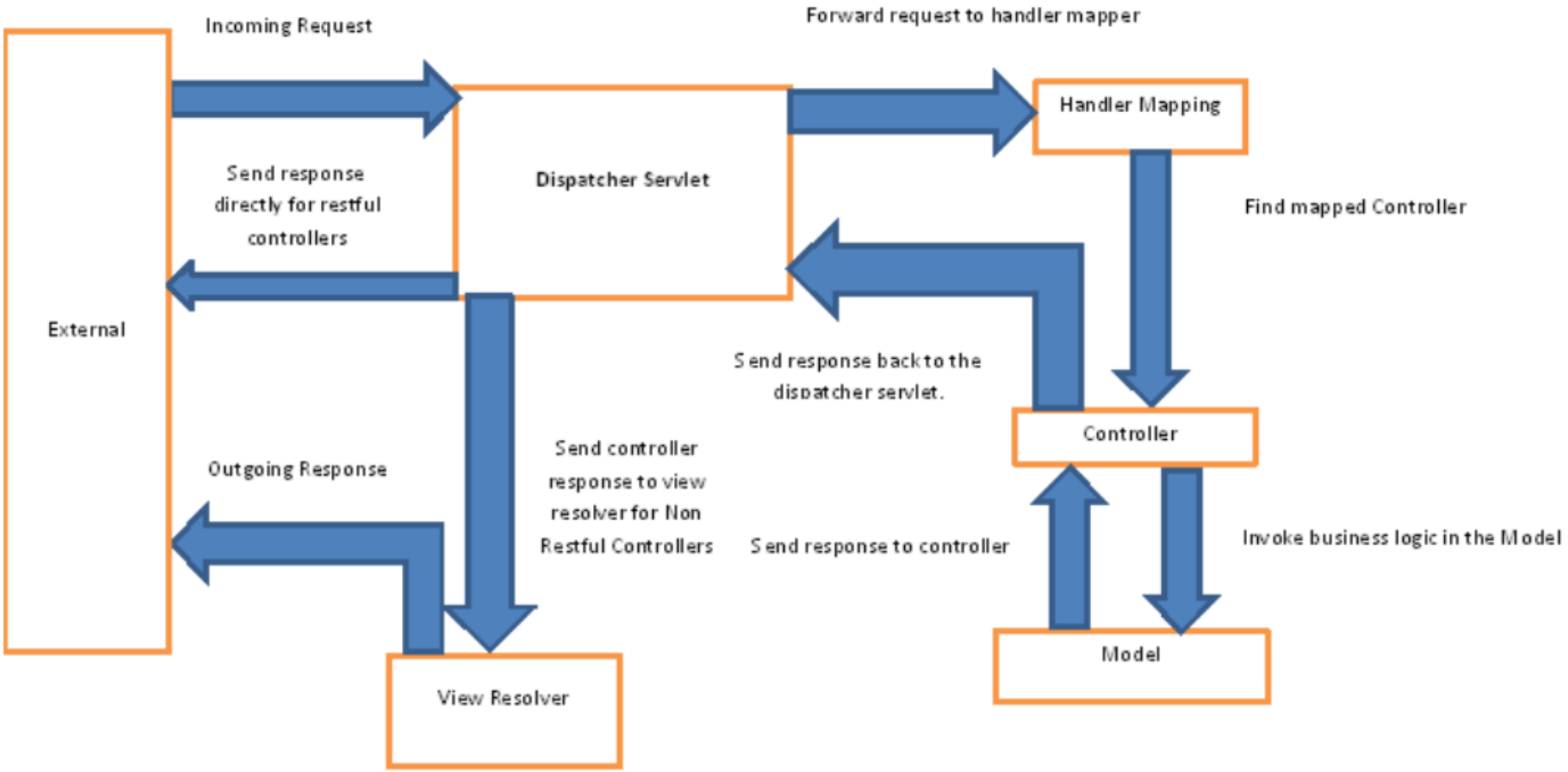
# Que fait un contrôleur ?

- Interprète et potentiellement transforme les requêtes
- Donne accès à la logique métier
- Détermine la vue ou le type de réponse
- Interprète les exceptions

# *Front Controller Pattern*

- Spring MVC utilise le patron *Front Controller* pour gérer le front avec les clients
- Ce rôle est joué par un objet particulier *Dispatcher Servlet*
- Ses missions sont :
  1. dispatcher les requêtes vers des composants *Delegate* gérés par Spring (i.e. *Request Mapping*)
  2. résoudre les vues (*View Resolution*)
  3. gérer les exceptions (*Exception Handling*)

# *Front Controller Pattern* -suite-



# A vos claviers

- Créer un projet Gradle (Java et Web) dans IntelliJ
- Ajouter les dépendances du projet dans Gradle :

```
dependencies {
  providedCompile "javax.servlet:javax.servlet-api:4.0.1"
  compile 'org.springframework:spring-context:5.2.9.RELEASE'
  compile 'org.springframework:spring-web:5.2.9.RELEASE'
  compile 'org.springframework:spring-webmvc:5.2.9.RELEASE'
  testCompile group: 'junit', name: 'junit', version: '4.13'
}
```

Ce script et les classes suivantes sont disponibles sur Moodle

# Configurer la *Dispatcher Servlet*

1. Définir la classe de configuration (`AppConfig`) :

```
@EnableWebMvc
@Configuration
@ComponentScan(basePackages = {"fr.umontpellier.fds.m2"})
public class AppConfig  implements WebMvcConfigurer {
 @Override
 public void addViewControllers(
         ViewControllerRegistry registry) {
  registry.addViewController("/").setViewName("index");
 }
 @Bean
 public ViewResolver viewResolver() {
  InternalResourceViewResolver bean =
    new InternalResourceViewResolver();
  bean.setPrefix("/WEB-INF/view/");
  bean.setSuffix(".jsp");
  return bean;
 }
}
```

# Configurer la `Dispatcher Servlet` -suite-

Dans la méthode `viewResolver` de la classe `AppConfig`, on a indiqué que :

- l'accès à la racine de l'application renvoie à la vue index
- les vues seront placées dans le dossier `WEB-INF/view` et que les noms de vues doivent être tout le temps suffixés par `.jsp`

# Configurer la `Dispatcher Servlet` -suite-

2. Définir une classe d'initialisation :

```
public class MainWebAppInitializer implements
    WebApplicationInitializer {
  @Override
  public void onStartup(final ServletContext sc) throws
    ServletException {
    // Charger la config de l'app Spring MVC
    AnnotationConfigWebApplicationContext ac = new
    AnnotationConfigWebApplicationContext();
    ac.register(AppConfig.class);

    // Creer et enregistrer la DispatcherServlet
    DispatcherServlet servlet = new DispatcherServlet(ac);
    ServletRegistration.Dynamic registration = sc.addServlet
    ("dispatcher", servlet);
    registration.setLoadOnStartup(1);
    registration.addMapping("/");
  }
}
```

# A vos claviers

- Ajouter un fichier `index.jsp` dans le dossier `webapp/WEB-INF/view`
- Mettre un lien dans cette page JSP qui pointe vers une vue (registration.jsp) qui fournit un formulaire HTML dans lequel l'on va saisir les informations concernant un utilisateur à inscrire sur un site Web
- On écrira plus tard la servlet `Controller` qui donne accès à la vue et traite les données du formulaire
- Pour déployer l'application, exécuter la tâche gretty/appRun, en allant dans la *tool window* "*gradle*" sur IntelliJ (en l'ouvrant depuis la barre latérale droite), puis `Tasks`, puis `gretty`, puis clique bouton-droit sur `appRun`, puis `Run`

  Pour le moment, la vue *registration* n'est pas accessible

# Annotations pour les contrôleurs de vues

- Annotations utilisées pour marquer les classes qui vont s'exécuter à la réception de requêtes HTTP
- Ces classes doivent être placées dans le package indiqué dans `@ComponentScan` dans la classe `AppConfig`, ou l'un de ses sous-packages (un sous-package `controller` par ex.)
- Dans ces classes, on déclare les méthodes qui vont recevoir chaque type de requête (Get, Post, ...) pour une route donnée
- Ces méthodes seront annotées `@GetMapping`, `@PostMapping` ou avec l'annotation générique `@RequestMapping`
- Les routes sont indiquées comme *value* dans ces annotations

# Exemple de contrôleur de vue

- Une classe RegistrationController

```java
package fr.umontpellier.fds.m2.controller;
// import ...
@Controller
public class RegistrationController {
  @GetMapping("registration")
  public String getRegistration() {
    return "registration";
  }
  @PostMapping("registration")
  public String addRegistration() {
    // ...
    return "registration";
  }
}
```

La première méthode permet de diriger vers la vue `registration`(.jsp) si l'utilisateur envoie une requête Get "${context.path}/registration"

# Annotations pour les contrôleurs de vues -suite-

- Les méthodes des contrôleurs peuvent déclarer des paramètres
- Ces paramètres peuvent être rattachés directement à des données transmises par les vues (champs dans un formulaire, par exemple)
- Ces paramètres sont annotés `@ModelAttribute`

# Exemple d'attribut de modèle

- Un objet (bean) de type `Registration` (qui fait partie du modèle est passé en paramètre)

```
@GetMapping("registration")
public String getRegistration(@ModelAttribute("
    registration") Registration registration) {
  return "registration";
}
@PostMapping("registration")
public String addRegistration(@ModelAttribute("
    registration") Registration registration) {
  System.out.println("Registration : "+registration.
    getName());
  return "process_registration";
}
```

# Lier les vues aux modèles

```
<%@ page contentType="text/html;charset=UTF-8"
         language="java" %>
<%@taglib prefix="form"
          uri="http://www.springframework.org/tags/form" %>
<html>
 <head><title>Signing up a user</title></head>
<body>
 <h2>Signing up a user</h2>
 <form:form modelAttribute="registration">
  <div>
  Name:
  <form:input path="name"/>
  ...
  </div>
  ...
 </div>
 </form:form>
</body>
</html>
```

# Lier les vues aux modèles -suite-

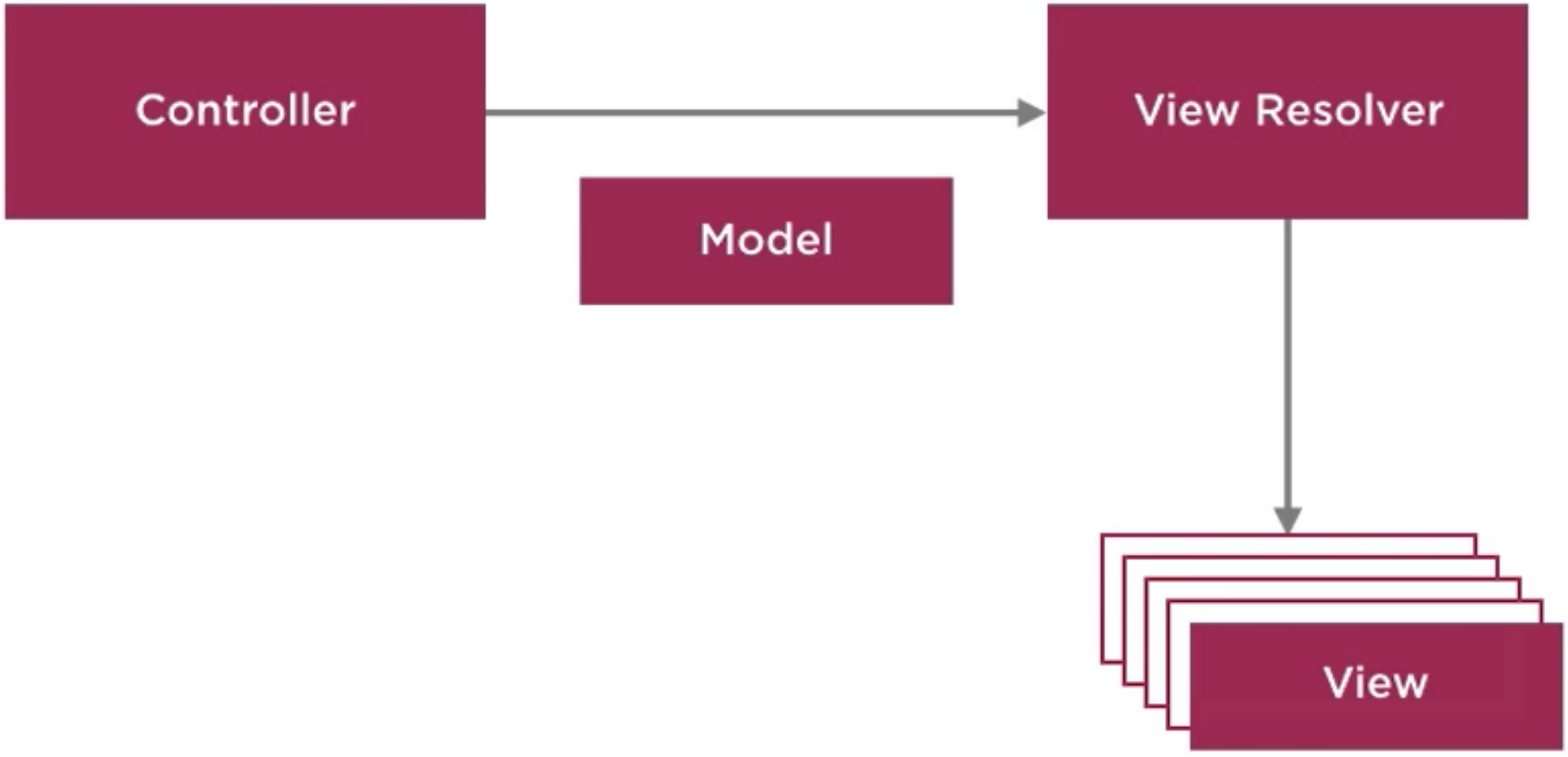
1. Déclarer dans la vue l'utilisation d'une librairie de balises (*taglib*) de Spring et lui donner un identifiant, qui servira comme préfixe
2. Utiliser ces balises (en préfixant les balises HTML du préfixe précédent – `form` dans l'exemple)
3. Déclarer un attribut `modelAttribute` ayant comme valeur un bean qui servira comme objet du modèle
4. Déclarer dans les éléments (champ d'un formulaire par exemple) le lien entre la valeur de cet élément et la valeur d'une propriété du bean (`name` dans l'exemple)

# Lier les vues aux modèles -suite-

- Dans l'exemple précédent, nous avions un bean dont l'id est `registration`
- Ajouter dans votre projet IntelliJ une classe Registration avec une propriété (attribut avec des accesseurs publics) `name`
- Tester votre application
- Ajouter d'autres champs dans le formulaire (email, ...)
- Compléter le code de la classe du bean
- Tester dans le contrôleur la bonne réception des données saisies dans le formulaire

# Plan du cours

# *View Resolver*




- Un objet utilisé par le contrôleur pour choisir les vues

# Dans les exemples précédents

- On a créé précédemment un *View Resolver* dans `AppConfig`

```
...
public class AppConfig  implements WebMvcConfigurer {
  ...
  @Bean
  public ViewResolver viewResolver() {
    InternalResourceViewResolver bean = new
    InternalResourceViewResolver();
    bean.setPrefix("/WEB-INF/view/");
    bean.setSuffix(".jsp");
    return bean;
  }
}
```

- Il choisit les vues dans le dossier `WEB-INF/view/` et ce sont des fichiers dont le nom est constitué du nom de la vue + ".jsp" (l'utilisateur ne verra pas .jsp)

# Résoudre les fichiers statiques

- Pour les fichiers statiques (HTML, CSS et JS, PDF, ...), on peut les placer dans un dossier WEB-INF/static/ (ensuite un sous-dossier par type de fichier : html/ js/ ...) et ensuite :

```
public void addResourceHandlers(ResourceHandlerRegistry
      registry) {
 registry.addResourceHandler("/*.html")
 .addResourceLocations("/WEB-INF/static/html/");
 registry.addResourceHandler("/images/**")
 .addResourceLocations("/WEB-INF/static/html/images/");
}
```

# Internationalisation *I18N* des vues

- Support pour d'autres langues/locales (anglais, espagnol, ...)
- Basée sur les intercepteurs (*Interceptors*)
- Procédure à suivre :
  1. Définir un *locale resolver* et indiquer la locale par défaut
  2. Définir et enregistrer un intercepteur de changement de locale
  3. Créer des fichiers .properties qui contiennent les messages dans différentes langues
  4. Remplacer dans les pages JSP le texte par des balises spéciales qui déclenchent ce système d'internationalisation

# Configuration de la locale

- Config de la locale par défaut dans un *Locale Resolver* à ajouter dans la classe `AppConfig` :

```
@Bean
public LocaleResolver localeResolver() {
 SessionLocaleResolver slr=new SessionLocaleResolver();
 slr.setDefaultLocale(Locale.FRANCE);
 return slr;
}
```

- Ajouter la source des messages (toujours dans la même classe) :

```
@Bean
public ReloadableResourceBundleMessageSource
    messageSource() {
  ReloadableResourceBundleMessageSource resource = new
    ReloadableResourceBundleMessageSource();
  resource.setBasename("WEB-INF/languages/messages");
  resource.setDefaultEncoding("UTF-8");
  return resource; }
```

# Configuration de la locale -suite-

- Config de l'intercepteur de changement de locale (`AppConfig`) :

```
@Bean
public LocaleChangeInterceptor localeChangeInterceptor
    () {
  LocaleChangeInterceptor lci = new
    LocaleChangeInterceptor();
  lci.setParamName("lang");
  return lci;
}
```

  Ici, on indique que la locale doit être lue dans le paramètre "`lang`" de la requête (`?lang=us`, par ex)

- Enregistrement de l'intercepteur :

```
public void addInterceptors(InterceptorRegistry
    registry) {
  registry.addInterceptor(localeChangeInterceptor());
}
```

# Implémenter les messages dans différentes locales

- Créer dans un dossier `WEB-INF/languages/` un fichier `messages.properties` avec le contenu suivant :

```
#labels
name=Nom
email=Courriel
```

  ça sera le fichier pour la locale par défaut (le français)

- Pour d'autres langues, créer, dans le même dossier, des fichiers `messages_us.properties` (pour l'anglais), `messages_es.properties` (pour l'espagnol), ...

# Utiliser les locales

- Dans une page JSP, déclarer la taglib :

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

- Remplacer le texte à internationaliser par des balises issues de la taglib
- Exemple : remplacer le label `Nom` par :

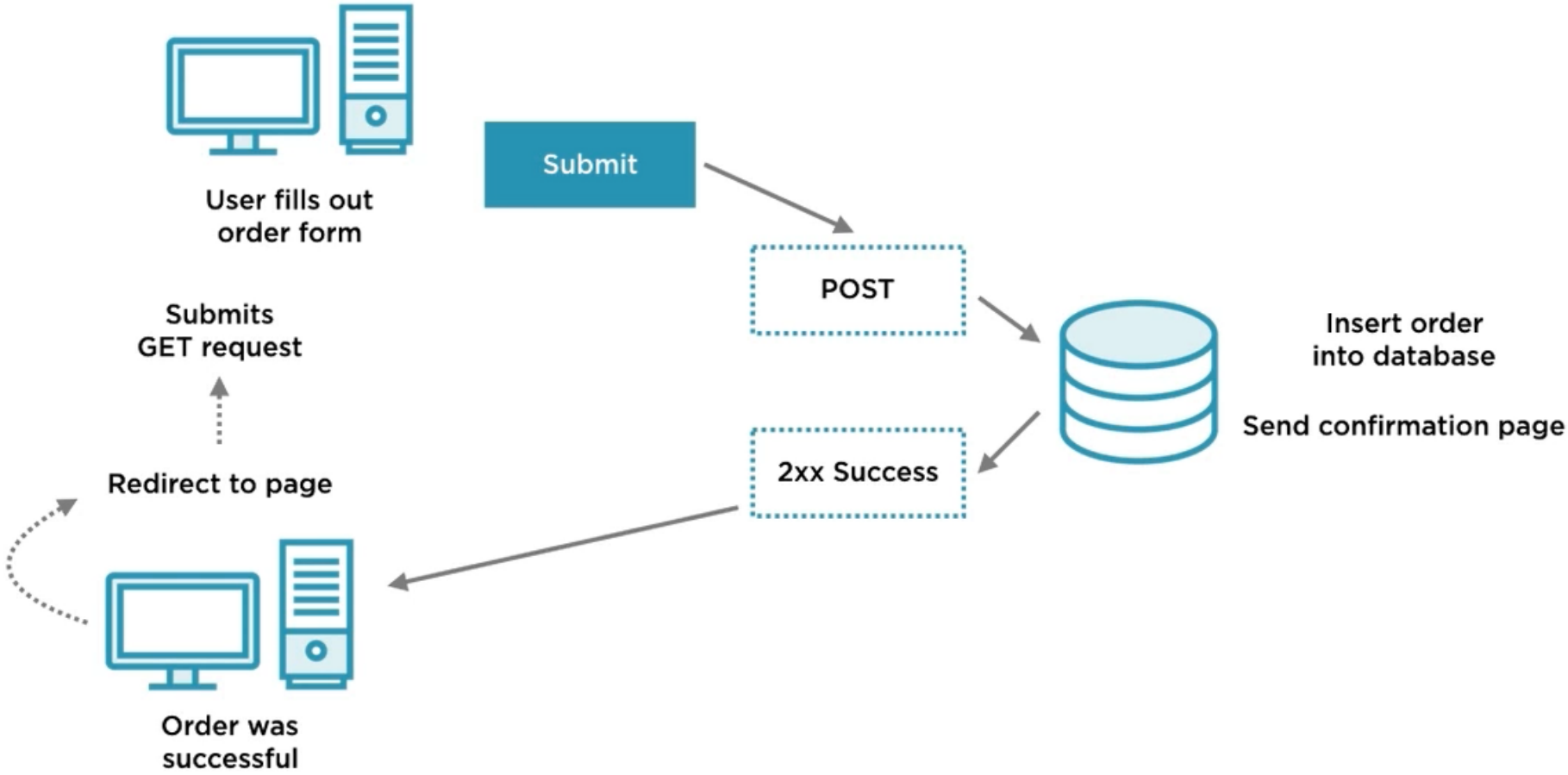
```
<spring:message code="name"/>
```

Selon la langue choisie par l'utilisateur, le bon fichier .properties sera lu pour obtenir la valeur du message "name"

# Quels solveurs de locales ?

- Spring fournit un certain nombre de solveurs concrets de locales
- Dans l'exemple précédent, nous avons utilisé un `SessionLocaleResolver`, qui utilise l'attribut `locale` dans la session utilisateur (locale valable toute une session)
- Autres solveurs :
  - `AcceptHeaderLocaleResolver` utilise l'entête HTTP de la requête "accept-language"
  - `FixedLocaleResolver` retourne une locale par défaut fixe
  - `CookieLocaleResolver` utilise une locale stockée dans un cookie

# PRG (*Post-Redirect-Get*) Pattern




- Objectif : après avoir traité un POST, on ne renvoie pas vers la vue d'origine ; on répond par une redirection pour que le navigateur fasse un Get de la vue (données sur la vue effacées)

# PRG (*Post-Redirect-Get*) Pattern -suite-

- Remplacer dans le contrôleur (le `return`)

```
@PostMapping("registration")
public String addRegistration(@ModelAttribute("
    registration") Registration registration) {
  System.out.println("Name: "+registration.getName());
  return "registration";
}
```

- par :

```
  @PostMapping("registration")
  public String addRegistration(@ModelAttribute("
    registration") Registration registration) {
    System.out.println("Name: "+registration.getName())
    ;
    return "redirect:registration";
  }
```

- Tester avant et après

# Plan du cours

1. Introduction à Spring MVC
2. Contrôleurs dans Spring MVC
3. Vues dans Spring MVC
4. Templates de vues *Thymeleaf*
5. Validation des beans des modèles
6. Services REST avec Spring MVC
7. Conclusion

# Pourquoi *Thymeleaf* ?

- Pour beaucoup de développeurs, avoir du JSP dans son application Web est considéré comme une dette technique
- On considère que JSP doit être remplacé par des technologies plus modernes de templating
- *Thymeleaf* est un framework de templating, qui s'intègre bien avec Spring
- De nos jours, on utilise ce genre de frameworks en combinaison avec des framework de front-end comme Angular, React, Vue, ...

# Dépendances à *Thymeleaf*

- Ajouter dans le build.gradle la dépendance :

```
compile "org.thymeleaf:thymeleaf-spring5:3.0.11.RELEASE"
```

- Tester le téléchargement des bibliothèques :
  - Sur IntelliJ, dans la tool-window Gradle, cliquer sur le bouton d'actualisation du projet
  - Aller ensuite dans l'explorateur du projet et ouvrir le dossier "Bibliothèques externes" ("*External Libraries*") pour vérifier si de nouveaux JAR Thymeleaf ont été ajoutés

# *Template Resolver*

- Dans le configurateur des contrôleurs et des solveurs de vues/locales, on ajoute une méthode qui retourne un bean de type `TemplateResolver` (différent d'un *View Resolver*) :

```
@Bean
public SpringResourceTemplateResolver templateResolver() {
  SpringResourceTemplateResolver templateResolver = new
    SpringResourceTemplateResolver();
  templateResolver.setApplicationContext(appContext);
  templateResolver.setPrefix("/WEB-INF/view/");
  templateResolver.setSuffix(".html");
  return templateResolver;
}
```

- Ajouter l'attribut (auto-injecté) `appContext` dans la classe :

```
@Autowired
private ApplicationContext appContext;
```

# *Template Engine*

- Moteur nécessaire pour produire des vues à partir de templates Thymeleaf
- Ajouter dans la même classe une méthode qui retourne un bean de type `TemplateEngine` :

```
@Bean
public SpringTemplateEngine templateEngine() {
  SpringTemplateEngine templateEngine =
    new SpringTemplateEngine();
  templateEngine.setTemplateResolver(
    templateResolver());
  templateEngine.setEnableSpringELCompiler(true);
  return templateEngine;
}
```

Noter la référence vers le *Template Resolver*

# *View Resolver*

- Ajouter un *View Resolver* dans la même classe que précédemment

```
@Bean
public ViewResolver thymeleafResolver() {
  ThymeleafViewResolver viewResolver = new
    ThymeleafViewResolver();
  viewResolver.setTemplateEngine(templateEngine());
  viewResolver.setOrder(0);
  return viewResolver;
}
```

- setOrder() doit indiquer une valeur inférieur à celle utilisée dans le view resolver de JSP (setOrder(1) pour le view resolver de JSP et setOrder(0) pour celui de Thymeleaf)
- Dans une vraie app, il vaut mieux enlever le view resolver de JSP (au lieu de faire cohabiter les 2)

# Définir des templates HTML avec Thymeleaf

- Créer une page HTML, (`home.html` par exemple) dans le dossier `WEB-INF/view` :

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
 <meta charset="UTF-8">
 <title>Accueil Thymeleaf</title>
</head>
<body>
 <h1>Accueil Thymeleaf</h1>
 <p th:text="${message}"></p>
</body>
</html>
```

- Noter le `xmlns:th` au début et l'attribut `th:text`
- D'où provient l'objet `message` utilisé ci-dessus ?

# Contrôleur pour les requêtes vers cette vue

- L'objet `message` dans cet exemple provient du modèle, mis à disposition par le contrôleur de requêtes HTTP vers cette vue :

```
@GetMapping("home")
public String getRegistration(Map<String, Object> model
    ) {
  model.put("message","Fantastic Thymeleaf!!!");
  return "home";
}
```

- C'est juste une chaîne de caractères ici

Plus de détails sur Thymeleaf ici : `https://www.thymeleaf.org/`
Comme vous pouvez le constater, les vues précédentes (JSP) ne sont plus accessibles (il y a une manip à faire pour faire cohabiter JSP et Thymeleaf, mais on ne le fera pas ici. Dans la suite, j'utilise JSP)

# Plan du cours

# Validation des beans

- Besoin de garantir des modèles valides dans les applications
- Même si certaines validations sont possibles en JavaScript côté client, il vaut mieux vérifier les données côté serveur (si jamais JS est désactivé sur le navigateur du client)
- Au départ, il y avait l'interface Validator, puis plusieurs JSR ont fait évoluer ce système de validation (dernière en date, JSR 380 pour Java8+ : API javax.validation)
- L'idée de base est de :
  1. contraindre les valeurs des propriétés de beans (objets du modèle)
  2. faire apparaître des messages d'erreur en cas de violation de ces contraintes

# Configuration de la validation

- Mise en place des dépendances (sur Gradle)

```
compile "org.hibernate.validator:hibernate-validator
    :6.0.2.Final"
```

  hibernate-validator est l'implémentation de référence de l'API de validation (javax.validation). Elle n'est pas liée à l'API de persistance de Hibernate

- Tester la présence de la bibliothèque dans votre projet

# Mise en place de la validation

- Annoter les attributs des objets du modèle
- Quelles annotations ?
  - `@NotEmpty`, `@NotNull`, `@AssertTrue`
  - ```
    @Size(min = 10, max = 200,
    message = "About Me must be between 10 and 200 chars")
    private String aboutMe;
    ```
  - ```
    @Email(message = "Email should be valid")
    private String email;
    ```
  - ```
    @Min(value=18, message="Age should not be < 18")
    @Max(value=150, message="Age should not be > 150")
    private int age;
    ```
- Ensuite, annoter avec `@Valid` les paramètres des méthodes des contrôleurs via lesquels transitent les objets du M
- Enfin, il faudra gérer les éventuelles erreurs
    ```

# Gestion des erreurs

- Dans une app Web, on va (dans une page JSP) :
  - 1. prévoir un bloc qui va apparaître en cas d'erreur (`<form:errors>` à l'intérieur de `<form>`)

```
<form:errors path="*" cssClass="errorblock"
             element="div"/>
```

    En ayant prévu une classe CSS `errorblock` (ou utiliser des classes de frameworks CSS comme Bootstrap)
  - Pour le faire avec Thymeleaf (ignorer la partie Spring Boot) :

    `https://spring.io/guides/gs/validating-form-input/`
- Dans l'exemple précédent, on va marquer :
  - l'attribut `email` de la classe `Registration` par l'annotation : `@Email(message="Email is not well-formed")`
  - et l'attribut `name` par l'annotation : `@NotEmpty(message="Name must not be empty")`

# Gestion des erreurs -suite-

- 2. affiner le contrôleur/solveur de vues :

```
@PostMapping("registration")
public String addRegistration(@Valid
        @ModelAttribute("registration")
        Registration registration,
        BindingResult result) {
  if(result.hasErrors()) {return "registration";}
  System.out.println(registration.getName());
  return "redirect:registration";
}
```

Noter ici :

- l'utilisation de l'annotation `@Valid`
- le second paramètre de la méthode, de type `BindingResult`
- le test `hasErrors()` et le retour à la vue (sans utilisation du patron PRG – `redirect:`)

# Personnaliser les messages d'erreur

- Il est possible d'utiliser l'internationalisation pour personnaliser les messages d'erreurs
- Dans les fichiers `messages.properties` vus précédemment, on va ajouter :
  NotEmpty.registration.name=Le champ nom ne doit pas \u00EAtre vide
  - Ici, `registration` correspond à l'id du bean concerné par l'annotation et `name` est le nom de la propriété du bean
  - le caractère 'ê' est remplacé ici par son code Unicode. Il existe des plugins Gradle (*native2ascii*, par ex), qui peuvent gérer les caractères avec accents dans des fichiers .properties

# Plan du cours

1. Introduction à Spring MVC
2. Contrôleurs dans Spring MVC
3. Vues dans Spring MVC
4. Templates de vues *Thymeleaf*
5. Validation des beans des modèles
6. Services REST avec Spring MVC
7. Conclusion

# Services REST

- Au lieu de servir des vues (pages HTML), une application Spring MVC peut servir des données, via des services REST (une alternative à JAX-RS de JEE)
- Par défaut, les services retournent des objets sérialisées comme données JSON
- Très simple à mettre en place :
  - Utiliser l'annotation `@RestController` pour marquer une classe contrôleur de services REST
  - Utiliser les annotations déjà vues (`@GetMapping`, ...) pour marquer les méthodes de cette classe (les *endpoints* du service)
  - Utiliser l'annotation `@RequestParam(value="...",defaultValue="...")` pour marquer les paramètres des méthodes
  - retourner des objets dans ces méthodes

# Exemple de service REST avec Spring

- Définir une classe (de beans) nommée User avec les propriétés `prenom`, `nom` et `age`
- Définir une classe `UserController`, annotée `@RestController`
- Y ajouter une méthode `getUser()` annotée `@GetMapping("/user")`

  Cette méthode crée un objet User et le retourne
- Ajouter 3 paramètres à cette méthode pour chaque propriété de beans User
  - Exemple de paramètre :
    @RequestParam(value="prenom",defaultValue="Elon")
    String prenom

# Mapping objets Java vers données JSON

- Dans le script de build de Gradle, ajouter les dépendances suivantes vers le mapper objets/JSON :

```
compile "com.fasterxml.jackson.core:jackson-core:2.11.2"
compile "com.fasterxml.jackson.core:jackson-databind:2.11.2"
```

- Tester le service REST précédent :
  `http://localhost:8080/votreprojet/user`
  `http://localhost:8080/votreprojet/user?prenom=Ian`

# Services REST de type POST, PUT, ...

- Pour les services REST de type Post, Put, Delete, ... le paramètre peut être un simple objet (User dans notre exemple)

```
@PostMapping("/user")
public User postUser(@RequestBody User user) {
  System.out.println("Prenom : "+user.getPrenom());
  return user;
}
```

- Cet objet est construit automatiquement par le mapper (prévoir un constructeur vide sans params) à partir des données de la requête HTTP
- Pour tester ce service, on peut utiliser `curl` ou des outils de test d'app Web comme *Postman*

Écrire maintenant une page HTML avec un script jQuery qui interroge le serveur pour récupérer les données d'un utilisateur (service GET précédent) et les affiche dans la page

# Plan du cours

# *Wrap-up*

- Développement d'app Web bien structurées, avec du typage statique à la Java pour plus de vérifications à la compilation
- Contrôleurs au coeur du framework Spring MVC
- Solveurs de vues et templating avec Thymeleaf
- Validation des modèles avec javax.validation
- Services REST très simples à mettre en place avec Spring MVC
- Aller plus loin en simplifiant plein de configurations avec Spring Boot (prochain cours)

# Références biblio

- Site Web de Spring : `https://spring.io/learn`
- Tutoriels sur Pluralsight et Baeldung
- Livres sur le sujet :
  `https://hackr.io/blog/spring-books`