

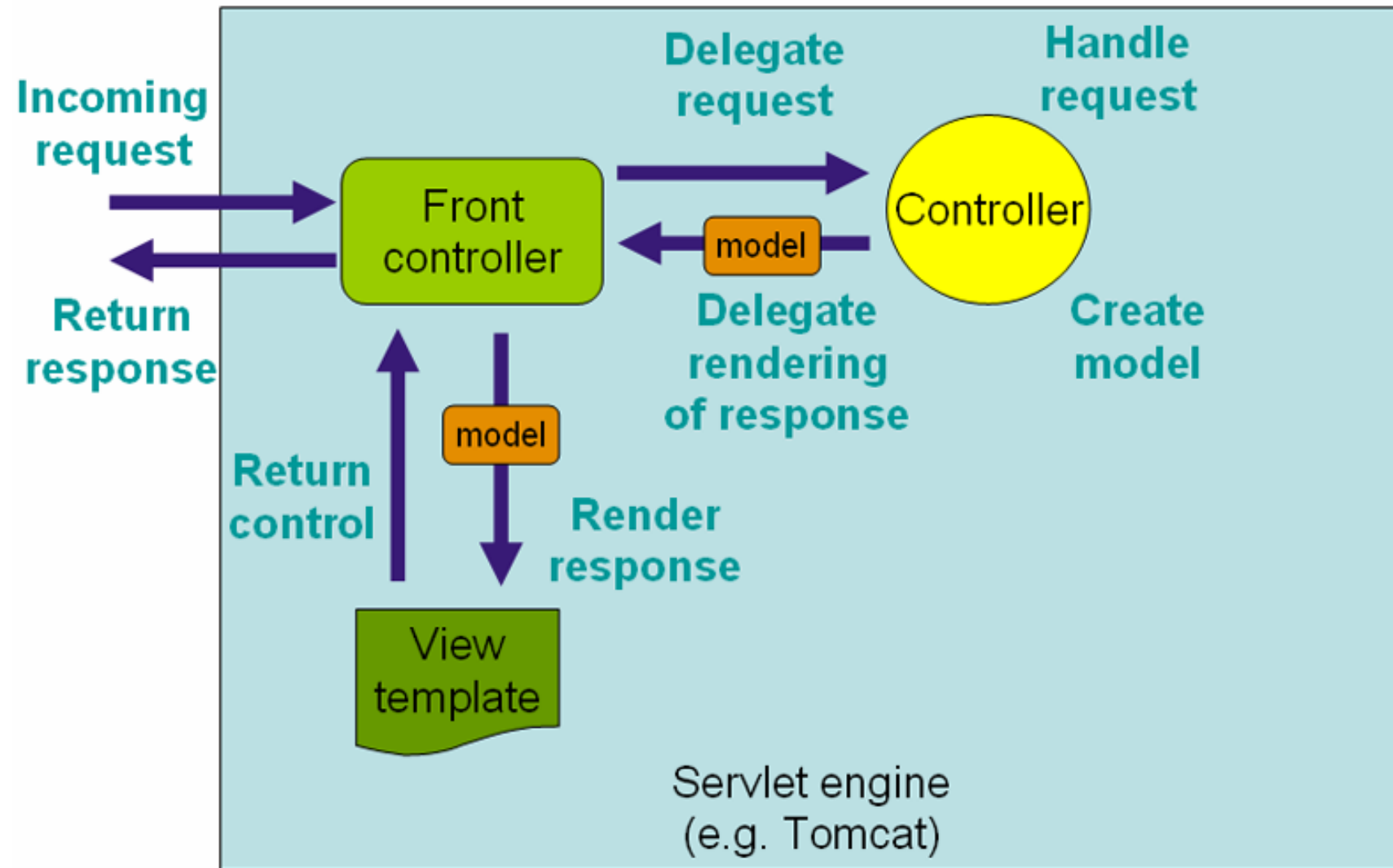
Formation Spring Framework

Spring MVC

Spring MVC

- Spring MVC est un framework de présentation implémentant le pattern MVC (Model View Controller)
- Il est flexible possible sur l'implémentation de ce pattern et n'impose pas de technologies pour la gestion des vues (ouverts à JSPs, FreeMarker, Velocity, XML, JasperReports, Excel et PDF) ou du modèle (POJO).
- Il repose sur un modèle simple Requête / Réponse en opposition avec un modèle événementiel plus complexe (ex : JSF).

Spring MVC



web.xml

```
<web-app>
```

```
<servlet>
```

```
  <servlet-name>dispatcher</servlet-name>
```

```
  <servlet-class>
```

```
    org.springframework.web.servlet.DispatcherServlet
```

```
  </servlet-class>
```

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

```
</servlet>
```

```
<servlet-mapping>
```

```
  <servlet-name>dispatcher</servlet-name>
```

```
  <url-pattern>/mvc</url-pattern>
```

```
</servlet-mapping>
```

```
</web-app>
```

Fichier de configuration Spring
par défaut **<nom-servlet>-servlet.xml**.
Ici : dispatcher-servlet.xml.

Contrôleur principal de Spring MVC

Front Contrôleur

- La servlet Spring officie comme Front Contrôleur (point d'entrée redirecteur).
- Par défaut, le fichier de définition Spring portant le nom de **<servlet-name>-servlet.xml** est chargé depuis le répertoire WEB-INF.
- Le nom de cette ressource est configurable en ajoutant :

```
<init-param>  
  <param-name>contextConfigLocation</param-name>  
  <param-value>/WEB-INF/config/web-application-config.xml</param-value>  
</init-param>
```

Contexte Général de Spring

- Le contexte général de Spring ajouté par "listener".
- Il s'agit d'un listener de type "ServletContextListener" (création et destruction de contexte de servlet).

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

Approche sans XML

- Spring MVC fournit la classe "org.springframework.web.WebApplicationInitializer" dont la méthode onStartUp est exécutée au démarrage de l'application.

```
public class WebAppInitializer implements WebApplicationInitializer {  
  
    private static final Logger LOG = Logger.getLogger(WebAppInitializer.class.getName());  
  
    @Override  
    public void onStartUp(ServletContext servletContext) throws ServletException {  
        LOG.log(Level.INFO, "démarrage du serveur");  
    }  
}
```

Approche sans XML

```
public class WebAppInitializer implements WebApplicationInitializer {

    private static final Logger LOG = Logger.getLogger(WebAppInitializer.class.getName());

    @Override
    public void onStartup(ServletContext servletContext) throws ServletException {
        LOG.log(Level.INFO, "demarrage du serveur");

        // Initialisation du contexte Spring
        AnnotationConfigWebApplicationContext webContext = new AnnotationConfigWebApplicationContext();
        webContext.register(PizzeriaSpringConfig.class);

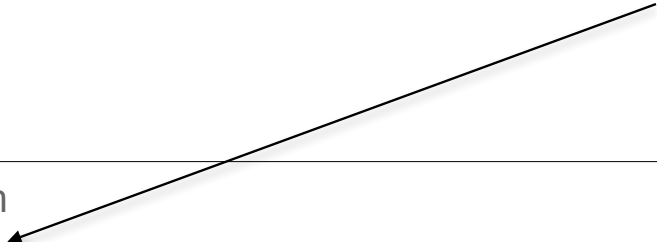
        /*
        <servlet>
            <servlet-name>dispatcher</servlet-name>
            <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
            <load-on-startup>1</load-on-startup>
        </servlet>
        */
        ServletRegistration.Dynamic dispatcher = servletContext.addServlet("dispatcher", new DispatcherServlet(webContext));
        dispatcher.setLoadOnStartup(1);

        /*
        <servlet-mapping>
            <servlet-name>dispatcher</servlet-name>
            <url-pattern>/api</url-pattern>
        </servlet-mapping>
        */
        dispatcher.addMapping("/api/*");

        /*
        <listener>
            <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
        </listener>
        */
        servletContext.addListener(new ContextLoaderListener(webContext));
    }
}
```


Activer Spring MVC

Activer Spring MVC
avec @EnableWebMvc



```
@Configuration
@EnableWebMvc
@ComponentScan("fr.pizzeria.spring.mvc.controller")
public class PizzeriaSpringConfig {
}
```


Contrôleur

- Un controller est une classe java annotée par `@Controller` et sert de point d'entrée à la coordination des traitements et du rendu logique.
- Le mapping entre urls et traitements se réalise par apposition de l'annotation `RequestMapping` sur les méthodes gestionnaires.

```
@Controller
@RequestMapping("/pizzas")
public class PizzaController {

    @RequestMapping(method = RequestMethod.GET)
    public String bonjour() {
        return "vuebonjour";
    }
}
```

Par défaut, le type retour désigne le nom d'une vue



Contrôleur

- L'annotation **@ResponseBody** permet de fournir directement la réponse à la requête sans passer par une vue.

```
@Controller
@RequestMapping("/pizzas")
public class PizzaController {

    @RequestMapping(method = RequestMethod.GET)
    @ResponseBody
    public String bonjour() {
        return "vuebonjour";
    }
}
```

Ici un navigateur affichera
"vuebonjour"

Signature des méthodes

- En paramètre

Request, Response et Session de l'API Servlet	
@PathVariable	Récupère une partie de l'url et la place dans le paramètre annoté.
@RequestParam	Récupère un paramètre de requête web (GET ou POST)
@RequestHeader	Récupère un header de requête web
@RequestBody	Accède au corps de la requête
Model	Classe contenant le modèle objet et permet son enrichissement avant transmission à la vue

Signature des méthodes

- Récupérer la requête, la réponse et la session de l'API Servlet

```
@Controller
@RequestMapping("/pizzas")
public class PizzaController {

    @RequestMapping(method = RequestMethod.GET)
    public String bonjour(HttpServletRequest request, HttpServletResponse response, HttpSession session) {
        // Du code
    }
}
```

Les objets en paramètres sont automatiquement valorisés



Signature des méthodes

- Récupérer une information dans le chemin de la requête

```
@Controller
@RequestMapping("/pizzas")
public class PizzaController {

    @RequestMapping(path =("/{nom}", method = RequestMethod.GET)
    public String bonjour(@PathVariable String nom) {
        // Du code
    }
}
```

Le nom de la variable correspond au nom du paramètre.

Autres exemples de paramètres d'entrée

```
@RequestMapping(method = RequestMethod.GET)
public String methode(@RequestParam("id") int id, Model model) {
    ...
}

@RequestMapping("/info")
public void info(@RequestHeader("Accept-Encoding") String encoding,
                 @RequestHeader("Keep-Alive") long keepAlive) {
    ...
}

@RequestMapping(value="/pizzas/{id}", method=RequestMethod.GET)
public void post(@PathVariable String id) {
    ...
}
```

Signature des méthodes

- En retour

ModelAndView	Encapsule le nom logique de la vue et son modèle.
Model	Encapsule le modèle. Retour sur la vue courante.
View	Retourne la vue préparée par l'application. Exemple : retour de documents générés.
String	Fournit le nom logique

REST / AJAX

- REST est un type d'architecture de web services défini par Roy Fielding.
- Il définit un protocole d'opérations web entre client et serveur collant au plus près au protocole de transport HTTP.
- Il comporte principalement les caractéristiques suivantes :
 - Stateless
 - Utilisation des verbes HTTP : GET , PUT, DELETE, HEAD, POST.
 - Les ressources sont identifiées par des URIs (surensemble contenant les URLs et URNs).
 - Protocole de communication du web : XML, JSON, ...
 - Hypermedia

REST / AJAX

- Spring MVC fournit les fonctionnalités utiles à l'implémentation d'un service RESTful.
 - L'attribut RequestMethod de RequestMapping permet de spécifier le verbe accepté
 - Les annotations de mapping de paramètres permettent de récupérer les identifiants de ressources depuis les URIs d'appel.
 - **@ResponseBody** permet de renvoyer la réponse au format HTTP grâce à un convertisseur.
 - Le convertisseur est un bean enregistré implémentant `HttpMessageConverter` qui définit la sérialisation / désérialisation HTTP. Spring fournit un jeu de classes convertisseurs prêt à l'usage

Controller

```
@RequestMapping(value = "/client/{clientId}",
method = RequestMethod.GET)
@ResponseBody
public Client findClient(@PathVariable int clientId) {
    return clientService.get(clientId);
}
```

@RestController = @Controller + @ResponseBody
@RestController se positionne sur une classe

REST / AJAX

- Une librairie de conversion doit être ajoutée au projet. Par exemple, Jackson.
- La directive `<annotation-driven/>` doit être ajoutée à la définition XML pour permettre la découverte des convertisseurs.
- Parmi les convertisseurs préconfigurés, on retrouve `MappingJacksonHttpMessageConverter`.
- Chaque convertisseur teste la classe à sérialiser pour déterminer s'il le prend en charge.

REST / AJAX

- Le format JSON du message retour est construit sur un mapping direct des champs du bean avec ceux du message.
- L'intégration avec les frameworks AJAX (jQuery, ...) est simplifiée par prise en charge des messages reçus JSON ou XML comme vu précédemment.
- En supplément, AJAX peut aussi envoyer au controller des messages dans le format choisi. Spring MVC les prend en charge également.

REST / AJAX

- L'annotation `RequestBody` informe du besoin de conversion sur le bean (XML ou JSON dans le cas AJAX).
- Le convertisseur correct sera choisi en fonction du header envoyé depuis le client : « `application/json` » ou « `application/xml` ».

```
public @ResponseBody Message create(@RequestBody Client client,  
    HttpServletResponse response) {  
    ....  
}
```

Gestion des vues

- Les classes d'implémentation de l'interface **ViewResolver** fournissent le mécanisme de sélection des vues selon un identifiant logique retourné par la méthode gestionnaire du controller.
- La plus courante d'utilisation avec les JSPs est **InternalResourceViewResolver** qui extrait la ressource depuis un chemin fabriqué avec le nom logique.
- Le nom logique est préfixé et suffixé pour fournir le chemin de la vue.

Exemple :

```
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">  
  <property name="prefix" value="/WEB-INF/views/" />  
  <property name="suffix" value=".jsp" />  
</bean>
```

Gestion des vues

- Le controller est chargé de sélectionner la vue consécutive à un traitement.
 - Pour cela , il renvoie un nom logique en fin de traitement.
- La nature polymorphique du type de retour implique différents moyens de sélection de la vue.

```
@RequestMapping("/helloWorld")
public ModelAndView helloWorld() {
    ModelAndView mav = new ModelAndView();
    mav.setViewName("helloWorld");
    mav.addObject("message", "Hello World!");
    return mav;
}

@RequestMapping("/helloWorld")
public String helloWorld() {
    return "helloWorld";
}
```


Gestion des vues

- Dans l'exemple précédent, la méthode **helloWorld** fournit à la vue les données par le biais du modèle transmis.
- La classe **ModelAndView** gère non seulement le nom logique de la vue mais aussi le modèle de la vue.
- Depuis la vue, l'attribut du modèle est directement référençable comme tout bean exposé à une page web.

Gestion des vues

```
@RequestMapping(method=RequestMethod.GET,value="list")
public ModelAndView listPeople() {
    ModelAndView mav = new ModelAndView();
    List<Person> people = personDao.getPeople();
    mav.addObject("people",people);
    mav.setViewName("list");
    return mav;
}
```

```
<c:forEach items="${people}" var="person">
    <a href="edit?id=${person.id}">
        ${person.id} - ${person.firstName} ${person.lastName}
    </a>
    <br/>
</c:forEach>
```

Création des formulaires

- Un formulaire Spring est une page web classique.
- Spring fournit un ensemble de taglibs pour faciliter la création de formulaire et le binding de données avec la partie serveur.

Création de formulaires

- Ecriture du gestionnaire d'envoi de formulaire

```
@Controller
@RequestMapping("/clientForm")
public class ClientFormController {
    ....
    @RequestMapping(method = RequestMethod.GET)
    public String setupForm(Model model) {
        // Création de l'objet du modèle.
        Client client = new Client();
        //Liaison du modèle et de l'objet.
        model.addAttribute("client", client);
        //Renvoi du nom logique de la vue formulaire.
        return "clientForm";
    }
    ...
}
```

Vue du formulaire

- Vue du formulaire :

- L'utilisation des taglibs est activée dans une JSP par la directive :

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

- Tous les éléments classiques des formulaires sont présents (checkbox, textarea, ...).
- Les gestionnaires d'événements sont aussi pris en compte (onclick, ...).
- En ajout, la liaison aux données et la gestion des messages d'erreurs.

Exemple de formulaire

```
<!-- pas besoin de définir une action, celle par défaut convient --%>
<form:form method="post" modelAttribute="client">
  <table>
    <tr>
      <td>Nom</td>
      <td><form:input path="nom" /></td>
    </tr>
    <tr>
      <td>Prenom</td>
      <td><form:input path="prenom" /></td>
    </tr>
  </table>
</form:form>
```

- L'attribut *modelAttribute* lie les données envoyées à un objet du modèle côté serveur.
- Les données retournées peupleront le bean identifié.

Envoi de formulaire

```
@RequestMapping(method = RequestMethod.POST)
    public String submitForm(@ModelAttribute("client")
Client client) {
    clientService.make(client);
    return "clientSuccess";
}
```

- L'objet *client* est peuplé selon les données de l'attribut du modèle nommé *client*.
- Cet attribut de modèle a été déclaré dans le formulaire.