

MODULE Spring

Spring MVC

Spring MVC

Plan du module Spring MVC

- Introduction
- Les patterns MVC et Contrôleur frontal
- Architecture Spring MVC
- Le contrôleur frontal
- Association URL / méthode de contrôleur
- Les contrôleurs
- La résolution de vue
- La génération de vue
- Autres points

Spring MVC

Introduction

Introduction

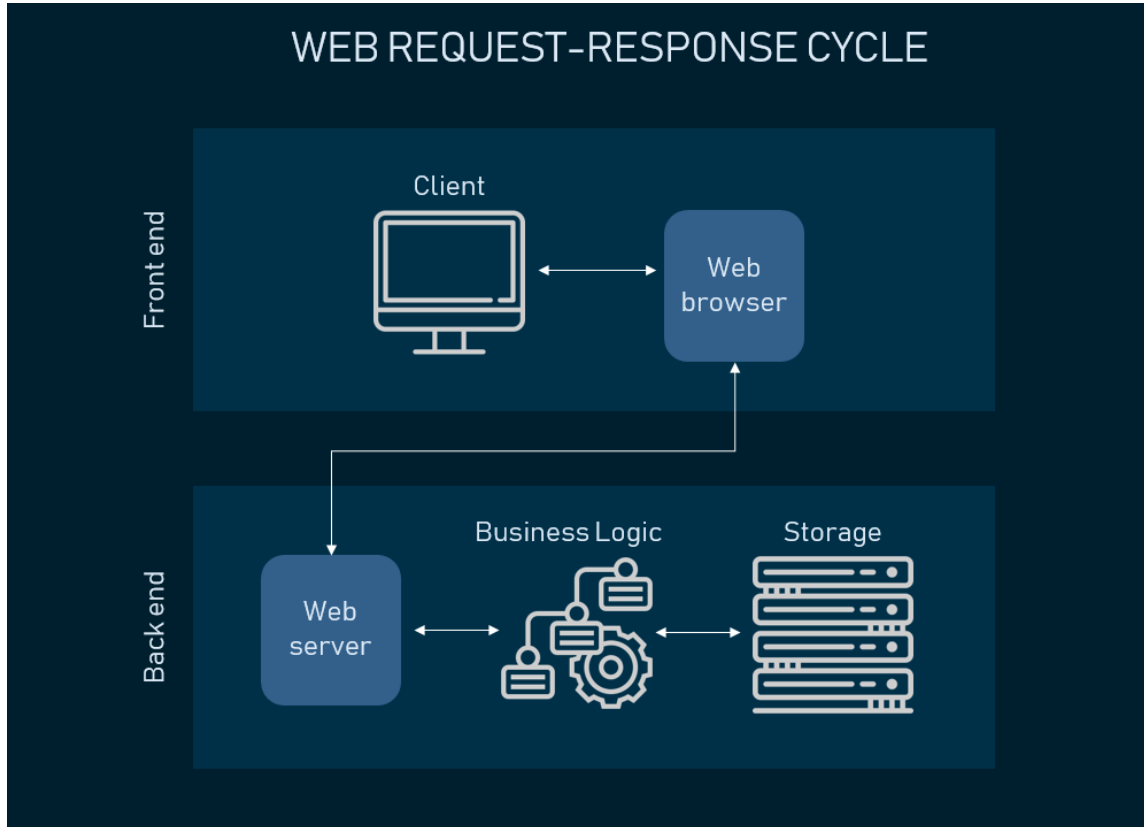
Qu'est-ce que l'architecture n-tier ?

Qu'est-ce qu'une application web ?

Qu'est-ce qu'une servlet ?

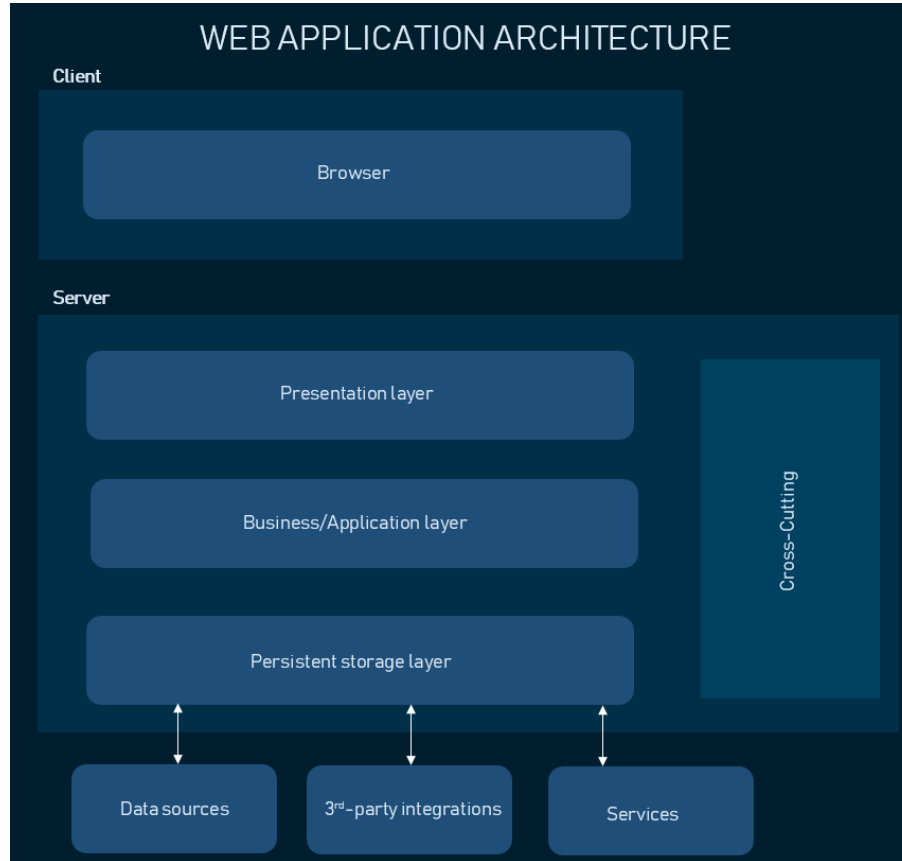


Introduction



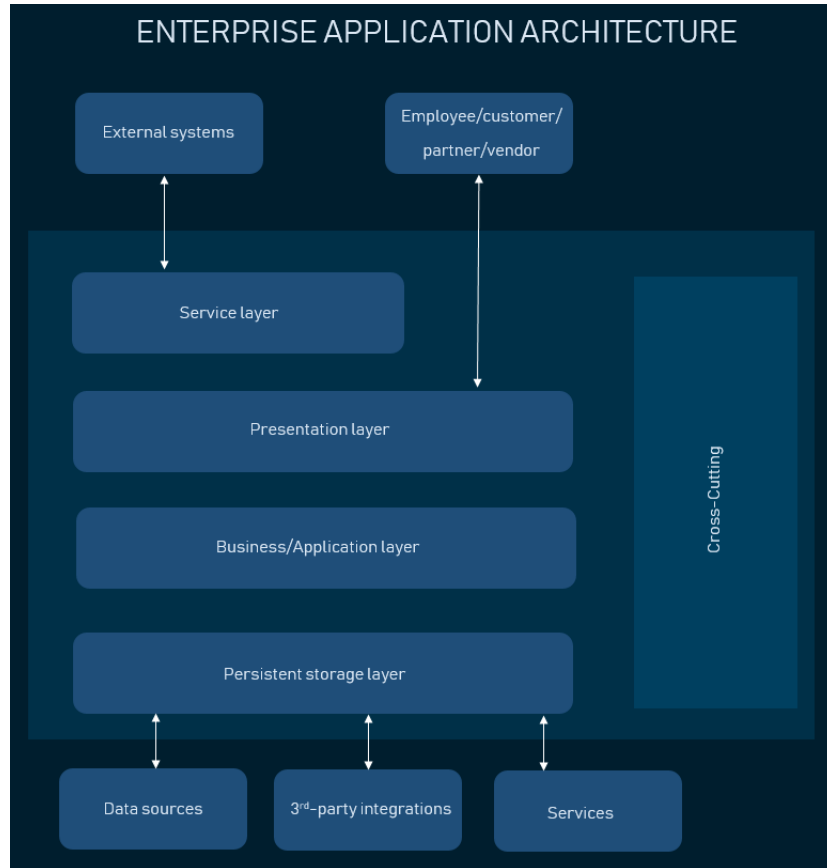
@Altexsoft

Introduction



@Altexsoft

Introduction



@Altexsoft

Spring MVC

Les patterns MVC et Contrôleur frontal

Les patterns MVC et Contrôleur frontal

Spring MVC est un ensemble de bibliothèques Spring permettant de faciliter le développement d'une application **web** générant des informations **visualisables par un utilisateur humain** (un site web, quoi!).

Autrement dit, pour Spring MVC :

- Le client est un navigateur web
- Le serveur est un serveur web (alias serveur HTTP)
- L'application génère (principalement) des pages HTML.

Les patterns MVC et Contrôleur frontal

De plus :

- Spring MVC ne s'intéresse qu'à la partie serveur
- Spring MVC est compatible d'un modèle client/serveur pur :
 - pas de notifications du serveur vers le client
⇒ ceci est l'objet de **Spring Reactive**

Les patterns MVC et Contrôleur frontal

Qu'est-ce que le design pattern (patron de conception) MVC ?



Les patterns MVC et Contrôleur frontal

Ce design pattern concerne principalement les applications avec une IHM.

Il vise à :

- séparer les différents types de traitement
- placer le code correspondant à ces types de traitement dans des modules séparés
- organiser la communication entre ces modules.

Les patterns MVC et Contrôleur frontal

Il distingue trois types de traitements :

- la gestion du modèle de données (les informations utiles ou « données métier » et leurs relations)
- les vues que l'application propose sur ce modèle de données (la façon dont les informations sont affichées)
- la prise en compte des requêtes de l'utilisateur et leur traitement.

Les patterns MVC et Contrôleur frontal

Les 3 parties qui résultent de cette analyse sont :

- le **Modèle** de données (Model)
- la **Vue** (View)
- le **Contrôleur** (Controller).

Les patterns MVC et Contrôleur frontal

Le modèle gère les données (il les valide, les enregistre, les récupère au besoin, etc.).

Ces données, a priori représentées par des beans, sont manipulées (créées, modifiées, supprimées, etc.) par le contrôleur sur requête de l'utilisateur.

Lorsque le modèle a été modifié, il informe la vue des nouvelles informations de manière à ce que l'IHM soit en phase avec le modèle. La vue utilise le modèle pour extraire les informations à afficher.

Les patterns MVC et Contrôleur frontal

Spring MVC propose d'organiser les applications web **grosso modo** selon ce pattern.

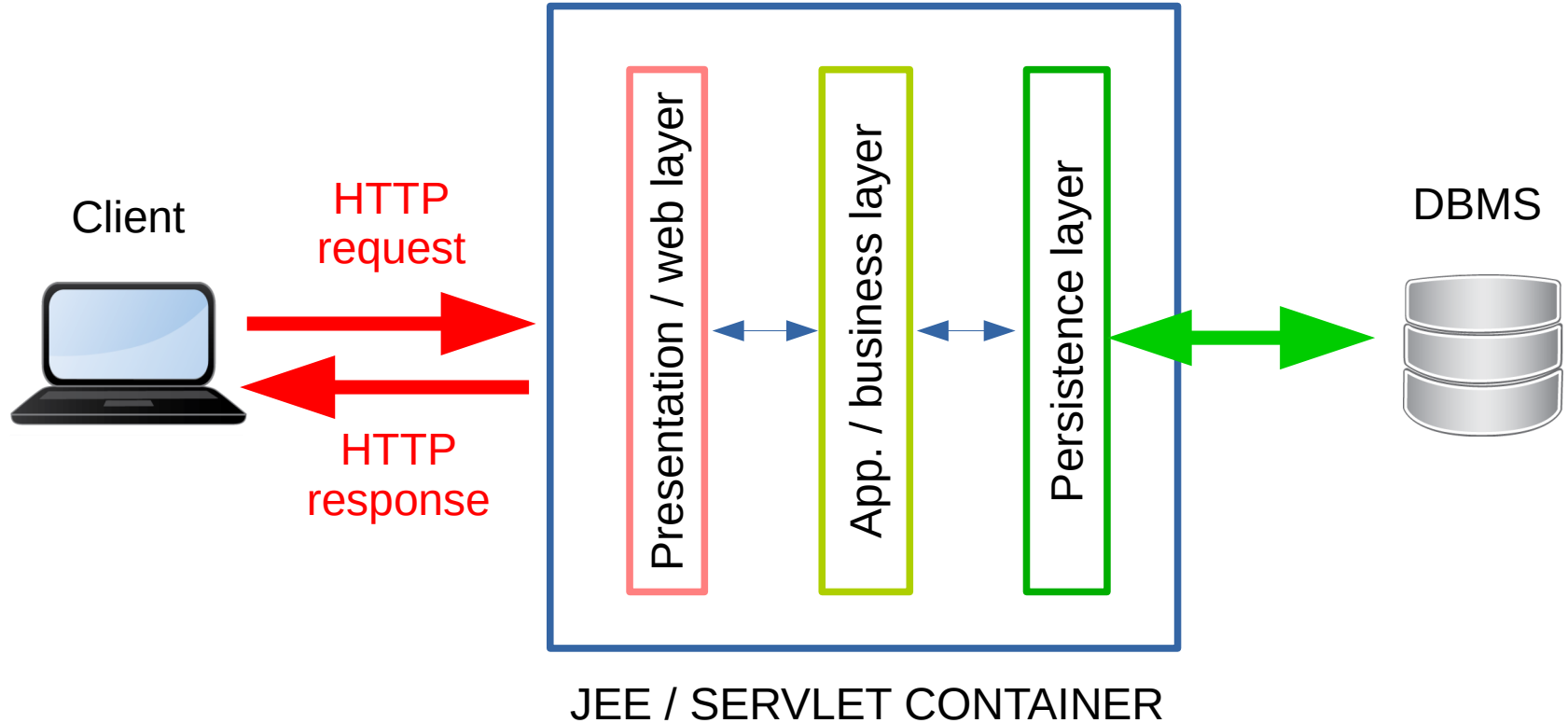
Il lui associe cependant un autre design pattern : le contrôleur frontal (« **front controller** »).

Ce pattern vise à centraliser toutes les tâches communes aux services fournis par une application web et à les invoquer en séquence. C'est une sorte d'ordonnanceur de tâches.

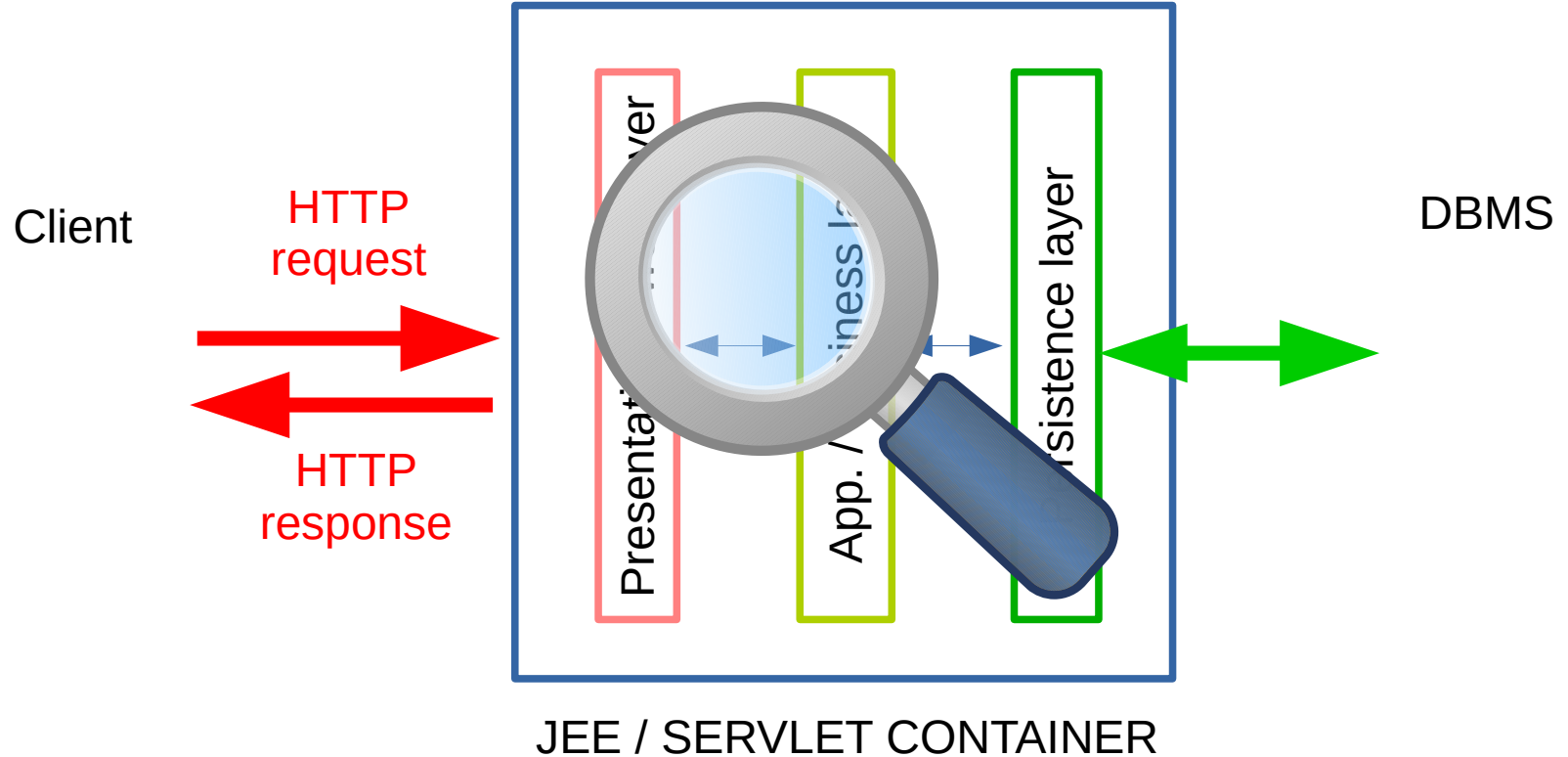
Spring MVC

Architecture Spring MVC

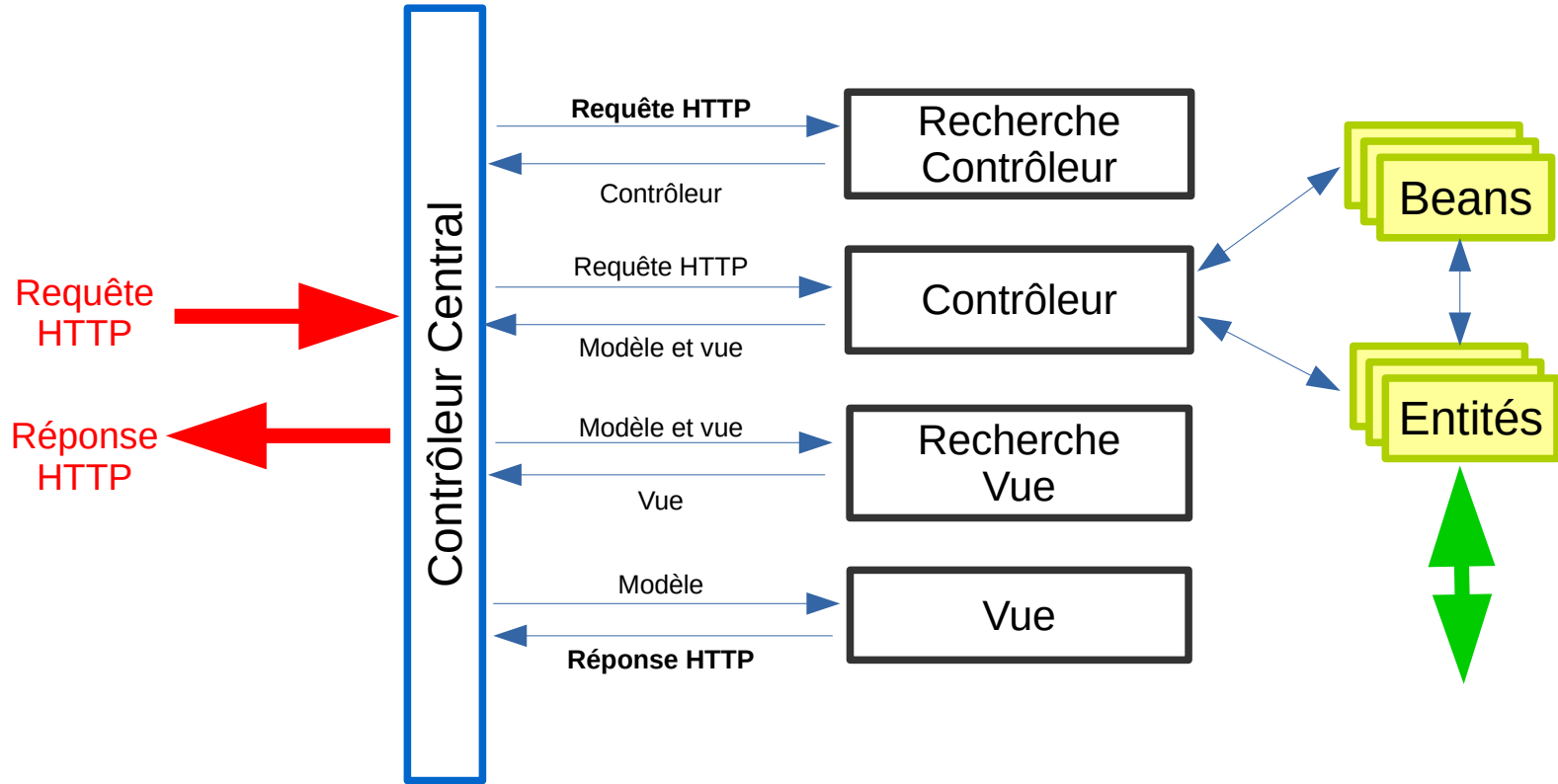
Architecture Spring MVC



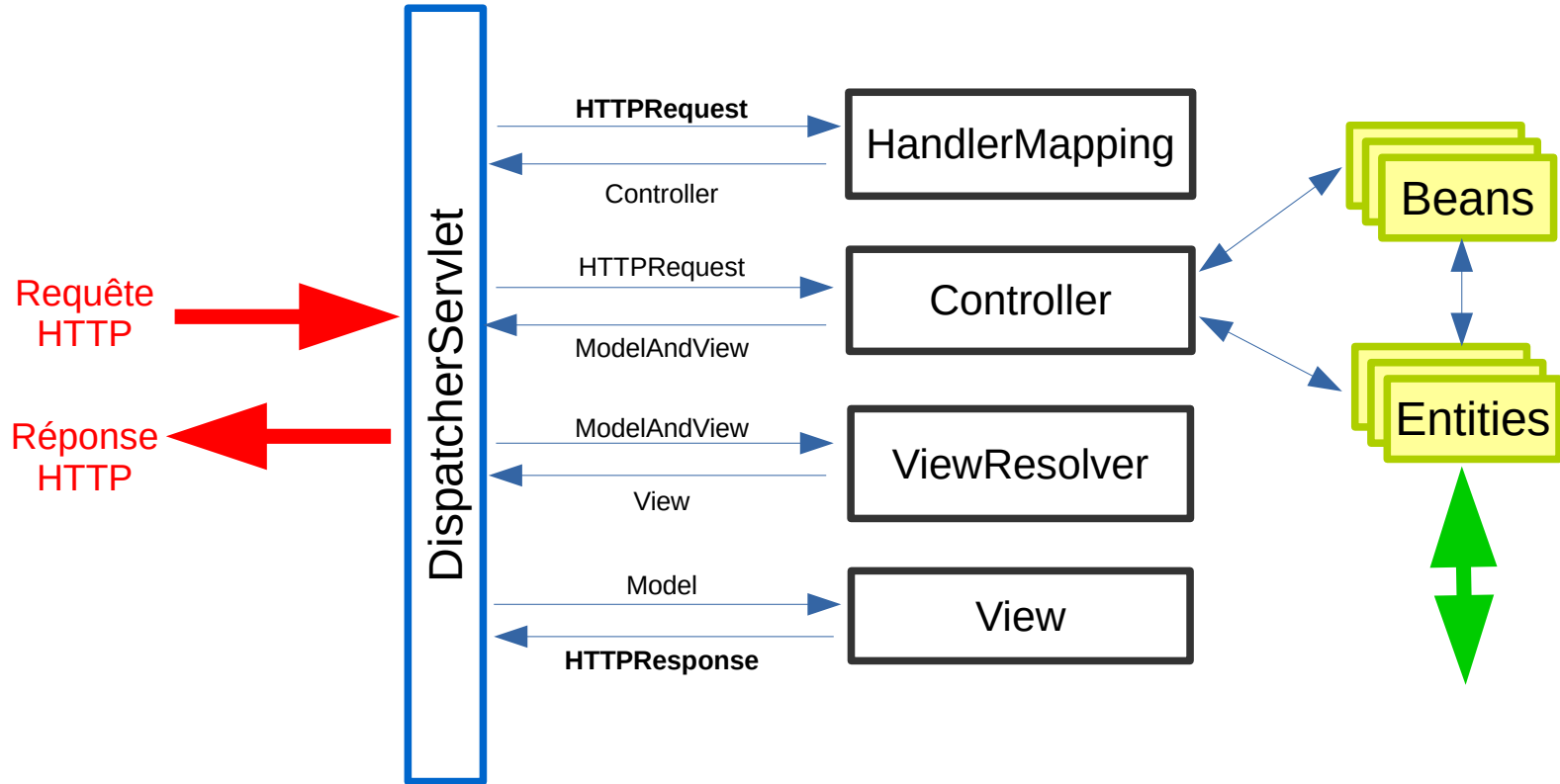
Architecture Spring MVC



Architecture Spring MVC



Architecture Spring MVC



Architecture Spring MVC

Quels sont les intérêts de cette architecture ?



Architecture Spring MVC

Intérêts de cette architecture d'application web :

- séparation des rôles (voir le **S** de SOLID...)
 - ==> réutilisation possible de composants métier ou DAO
- pas de contrainte sur le choix de la technologie d'affichage des pages
 - ==> grâce à la technique de résolution de vues

Architecture Spring MVC

Intérêts de l'utilisation de Spring dans ce contexte :

- fourniture de nombreux services “gratuits”
- pas besoin d'un serveur d'application JEE (lourd)
- configurabilité élevée
- auto-composition des services (IoC container)

Architecture Spring MVC

Rappel : une application Spring, qu'elle soit du type MVC ou pas, s'exécute dans un **contexte applicatif** qui représente l'IoC container de Spring.

Dans le cas d'une application web Spring MVC, ce contexte applicatif est en fait plus précisément une implémentation de l'interface `WebApplicationContext`.

Cette interface est une extension de `ApplicationContext`. Elle permet d'accéder au contexte de servlets (`ServletContext`, voir cours sur les servlets) dans lequel l'application s'exécute.

Architecture Spring MVC

Ce contexte applicatif gère les beans qui y sont déclarés, y compris les beans représentant des objets d'une application web : contrôleur central, contrôleurs, résolveurs de vues, etc.

Architecture Spring MVC

Ainsi, une classe de mapping entre une URL et un contrôleur peut être considérée comme une **dépendance du contrôleur central**.

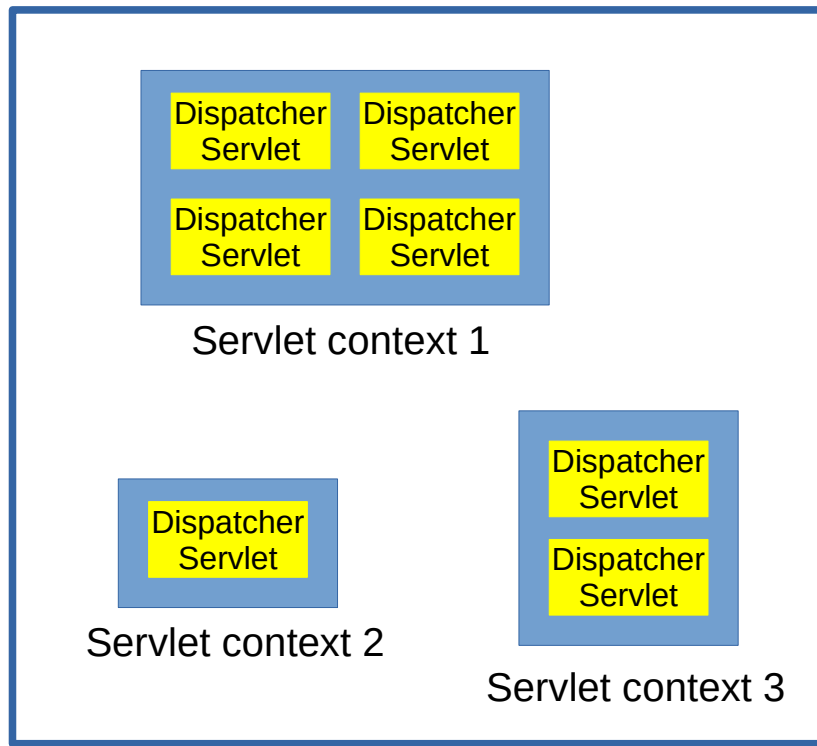
Même chose pour un contrôleur, ou une classe de résolution de vues, qui sont de facto des **dépendances du contrôleur central**.

Les concepts vus précédemment peuvent donc être mis en oeuvre dans le contexte d'une application de type Spring MVC.

Architecture Spring MVC

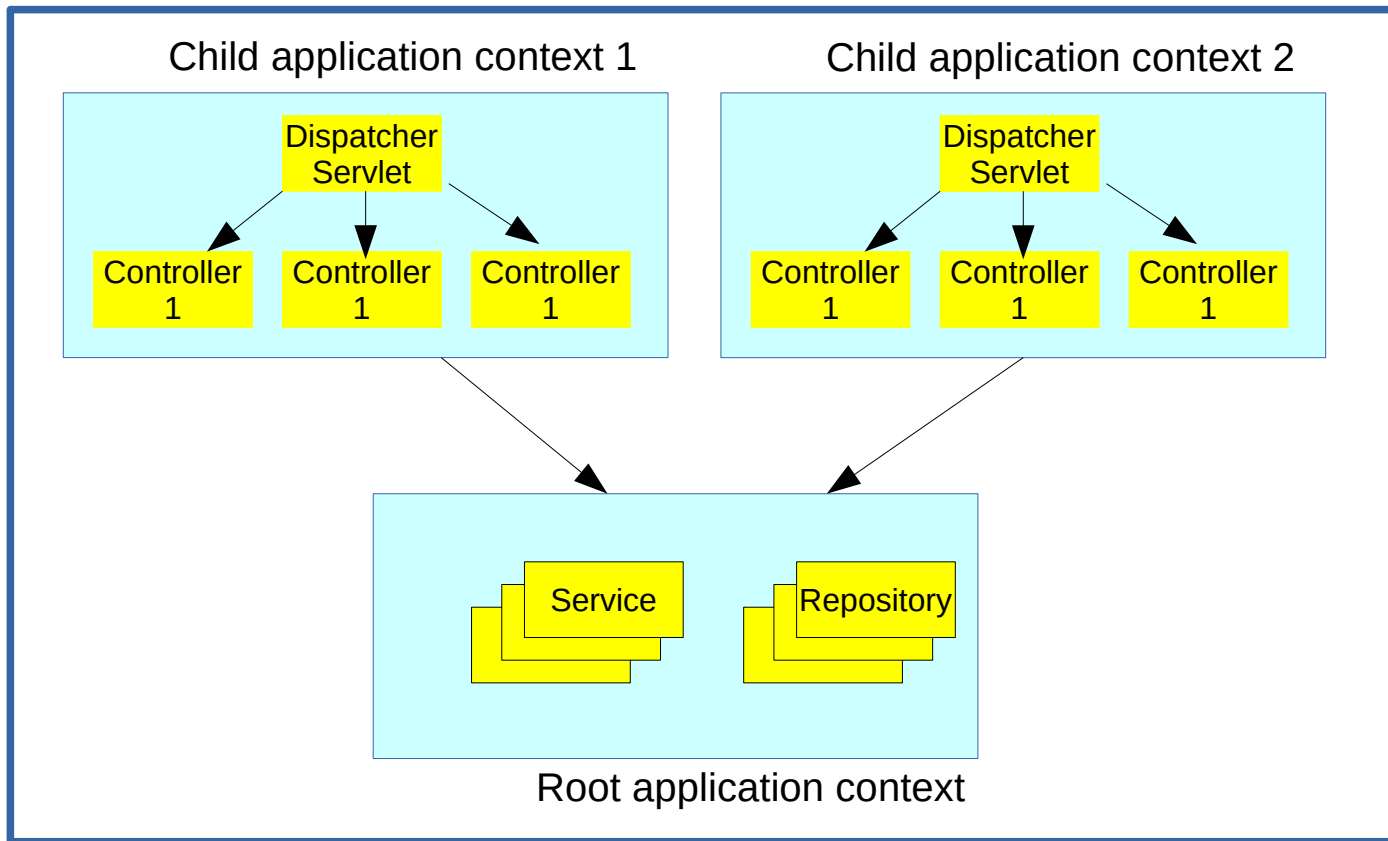
Mais un contrôleur central est aussi une servlet, il s'exécute dans un contexte de servlet (`ServletContext`), de par la spécification des servlets.

Architecture Spring MVC

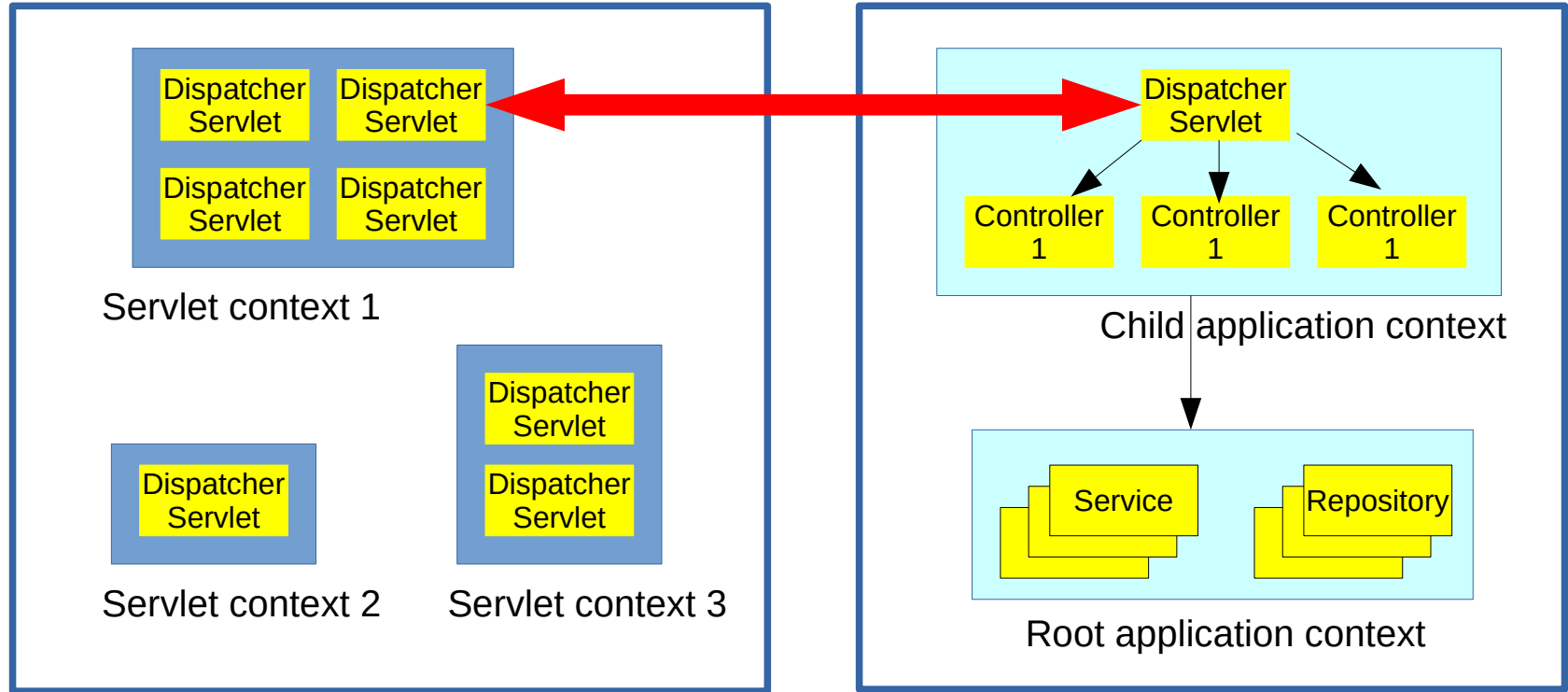


JEE / SERVLET CONTAINER

Architecture Spring MVC



Architecture Spring MVC



JEE / SERVLET CONTAINER

SPRING IOC CONTAINER

Architecture Spring MVC

Le contexte applicatif racine est géré par un listener du type `org.springframework.web.context.ContextLoaderListener`, membre du module `spring-web`.

Ce listener charge par défaut le contexte applicatif à partir du fichier `/WEB-INF/applicationContext.xml`.

Il peut être configuré lui-même à partir du fichier de configuration de la webapp (`web.xml`, voir cours sur les servlets).

Architecture Spring MVC

```
<web-app>
```

```
  <context-param>
```

```
    <param-name>contextConfigLocation</param-name>
```

```
    <param-value>/WEB-INF/mon-app.xml</param-value>
```

```
  </context-param>
```

```
  ...
```

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

Fichier de définition
du contexte IoC

Architecture Spring MVC

Le contexte applicatif peut aussi être configuré de manière programmatique, depuis la version 3 de l'API Servlet.

Ceci est réalisé au moyen de l'interface `WebApplicationInitializer`. La méthode `onStartup()` de cette interface est l'endroit où on peut créer le contexte applicatif.

Architecture Spring MVC

```
public class monInitialiseur implements
WebApplicationInitializer {

    @Override

    public void onStartup(ServletContext servletContext) {

        AnnotationConfigWebApplicationContext contexte = new
AnnotationConfigWebApplicationContext();

        contexte.register(AppConfig.class); // Classe Java de
        configuration

        contexte.refresh();

        ...

    }
```

Architecture Spring MVC

Squelette de classe de configuration.

@Configuration

```
public class AppConfig {  
    ...  
}
```

Architecture Spring MVC

On peut aussi spécifier le contexte applicatif depuis le fichier `web.xml` tout en utilisant la configuration basée sur Java pour le configurer.

```
<context-param>
```

```
    <param-name>contextClass</param-name>
```

```
    <param-value>
```

```
    org.springframework.web.context.support.AnnotationConfig  
    WebApplicationContext</param-value>
```

```
</context-param>
```

Architecture Spring MVC

Et il existe beaucoup d'autres manières de préciser le contexte applicatif...

Spring MVC

Le contrôleur frontal

Le contrôleur frontal

La classe `DispatcherServlet`

Une application web peut être composée de plusieurs servlets. Dans le framework Spring, c'est la même chose : une application Spring MVC peut être composée de plusieurs `DispatcherServlet`.

Comme toujours avec Spring, la configuration d'une `DispatcherServlet` peut être faite automatiquement ou pas, et ce de différentes manières : par programme, par fichier de configuration XML ou par annotation.

Le contrôleur frontal

En tant que servlet, le contrôleur frontal doit être connu et géré par le container de servlets. Ceci peut être réalisé de manière programmatique au moyen de l'interface `WebApplicationInitializer`.

Le contrôleur frontal peut ainsi être créé dans l'implémentation de la méthode `onStartup(ServletContext)` de cette interface.

Le contrôleur frontal

Comme vu ci-dessus, le contrôleur frontal s'exécute dans un contexte applicatif. Ce contexte applicatif lui est passé à sa création.

La méthode `onStartup()` de cette interface est donc l'endroit où on peut créer ce contexte applicatif.

Exemple ci-dessous.

Le contrôleur frontal

```
public class monInitialiseur implements
WebApplicationInitializer {

    @Override

    public void onStartup(ServletContext servletContext) {

        AnnotationConfigWebApplicationContext contexte =
        new AnnotationConfigWebApplicationContext();

        contexte.register(AppConfig.class); // Classe Java
        de configuration

        contexte.refresh();
    }
}
```

Le contrôleur frontal

```
DispatcherServlet contrôleurFrontal = new  
DispatcherServlet(contexte);
```

```
ServletRegistration.Dynamic controleur =  
servletContext.addServlet("contrôleurCentral",  
contrôleurFrontal);
```

```
contrôleur.setLoadOnStartup(1);
```

```
contrôleur.addMapping("/appli/*");
```

```
}
```

```
}
```

Le contrôleur frontal

Il existe d'autres manières de configurer par programme un contrôleur frontal, par exemple en étendant les classe abstraites `AbstractDispatcherServletInitializer` ou `AbstractAnnotationConfigDispatcherServletInitializer`.

Le contrôleur frontal peut aussi être créé au travers du fichier XML de définition de l'application web (`web.xml`).

Exemple ci-dessous.

Le contrôleur frontal

Fichier de définition
du contexte IoC

```
<web-app>
```

```
  <context-param>
```

```
    <param-name>contextConfigLocation</param-name>
```

```
    <param-value>/WEB-INF/mon-app.xml</param-value>
```

```
  </context-param>
```

```
  <servlet>
```

```
    <servlet-name>contrôleurFrontal</servlet-name>
```

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

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

```
  </servlet>
```

Le contrôleur frontal

```
<servlet-mapping>
```

```
  <servlet-name>contrôleurFrontal</servlet-name>
```

```
  <url-pattern>/appli/*</url-pattern>
```

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

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

Le contrôleur frontal

Il existe encore une autre manière, plus “transparente”, de définir le contrôleur frontal et le contexte applicatif : en utilisant **Spring Boot Starter**.

Spring Boot Starter est un outil permettant (entre autres) de créer (entre autres) des applications web Spring “**self-contained**” :

- Elles intègrent un container de servlets et n'ont donc pas besoin d'être placées dans un container de servlet externe
- Elles peuvent être **packagées** (intégrées) sous forme d'un JAR exécutable.

Le contrôleur frontal

Elles peuvent bien sûr être aussi packagées sous forme d'un WAR.

De plus, elles nécessitent **très peu de configuration**, que ce soit externe (par fichier XML de définition de contexte ou d'application web) ou interne (par configuration Java) : elles sont conçues pour rechercher leur configuration automatiquement, principalement dans le code Java lui-même ("Java-based configuration").

Le **système de logging** est intégré automatiquement (SLF4J, utilisant log4j ou logback), et l'**environnement de test** (JUnit typiquement) est fourni dès le départ.

Le contrôleur frontal

Une application Spring Boot est définie par une classe annotée par **@SpringBootApplication**.

Cette méta-annotation regroupe 3 annotations :

- @SpringBootConfiguration
==> principale classe de configuration basée sur Java
- @EnableAutoConfiguration
==> configuration automatique de l'application
- @ComponentScan
==> le package contenant l'application est auto-scanné

Le contrôleur frontal

L'annotation `@EnableAutoConfiguration` fait beaucoup de choses, car elle choisit la configuration à utiliser en fonction des fichiers JAR qui se trouvent dans le classpath.

Exemple : si le fichier `tomcat-embedded.jar` est dans le classpath, alors un container Tomcat sera **automatiquement fourni** au travers d'une fabrique `TomcatServletWebServerFactory`.

Le contrôleur frontal

La dépendance Maven à indiquer pour définir une application web Spring Boot Starter est la suivante.

```
<dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-web</artifactId>  
</dependency>
```

Le contrôleur frontal

Le squelette d'une application web Spring Boot est :

```
@SpringBootApplication
```

```
public class SpringBootAppl {
```

```
    public static void main(String[] arguments) {
```

```
        SpringApplication.run(SpringBootAppl.class,  
                                args);
```

```
    }
```

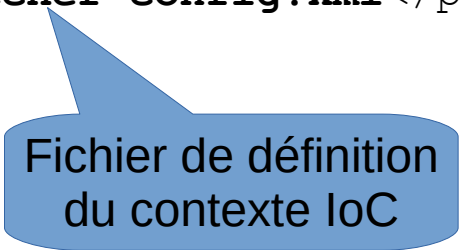
```
}
```

Le contrôleur frontal

En synthèse, pour définir et configurer le contrôleur frontal, il existe plusieurs possibilités :

- Utiliser un fichier `web.xml` classique pour définir la classe (`DispatcherServlet`) et son contexte applicatif :

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



Fichier de définition
du contexte IoC

Le contrôleur frontal

- Utiliser la configuration basée sur Java

- Création d'un contexte applicatif adapté

```
AnnotationConfigWebApplicationContext contexte =  
new AnnotationConfigWebApplicationContext();
```

- Enregistrement d'une classe de configuration dans ce contexte

```
contexte.register(AppConfig.class);
```

- (Ou bien scan d'un package complet pour recherches les classes de configuration contenues)

```
contexte.setConfigLocation("mon.package.a.moi");
```

Le contrôleur frontal

- Ajout d'une classe de chargement du contexte applicatif dans le contexte web

```
container.addListener(new  
ContextLoaderListener(contexte));
```

Contexte IoC défini
précédemment

- Création du contrôleur frontal lui-même :

```
ServletRegistration.Dynamic controleurFrontal =  
container.addServlet("dispatcher", new  
DispatcherServlet(contexte));
```

Container
de servlets

- Configuration du contrôleur frontal lui-même

```
controleurFrontal.setLoadOnStartup(1);  
controleurFrontal.addMapping();
```


Le contrôleur frontal

```
public class MyWebAppInitializer implements  
WebApplicationInitializer {
```

```
    @Override
```

```
    public void onStartup(ServletContext container) {
```

```
        AnnotationConfigWebApplicationContext contexte
```

```
        = new AnnotationConfigWebApplicationContext();
```

```
        contexte.setConfigLocation("mon.package.a.moi");
```

```
        container.addListener(new ContextLoaderListener(contexte));
```



Container
de servlets

Le contrôleur frontal

```
ServletRegistration.Dynamic contrôleurFrontal =  
container.addServlet("dispatcher", new  
DispatcherServlet(contexte));  
  
    contrôleurFrontal.setLoadOnStartup(1);  
  
    contrôleurFrontal.addMapping("/");  
  
}
```



Contrôleur
frontal

Le contrôleur frontal

- Troisième possibilité : utiliser Spring Boot et laisser l'outil créer des objets selon le contexte, s'auto-configurer, etc.
- Quatrième possibilité (eh oui, c'est Spring...) : panacher ces différentes manières de faire.
 - Exemple page suivante, où on mixe la configuration basée sur Java avec l'utilisation d'un bon vieux fichier XML pour définir le contexte applicatif

Le contrôleur frontal

```
public class MyWebAppInitializer implements WebApplicationInitializer {  
    @Override  
    public void onStartup(ServletContext container) {  
        XmlWebApplicationContext contexte = new XmlWebApplicationContext();  
        contexte.setConfigLocation("/WEB-INF/spring/app-config.xml");  
  
        ServletRegistration.Dynamic contrôleurFrontal =  
        container.addServlet("dispatcher", new DispatcherServlet(contexte);  
  
        contrôleurFrontal.setLoadOnStartup(1);  
  
        contrôleurFrontal.addMapping("/");  
    }  
}
```

Container
de servlets

Association URL / méthode de contrôleur

La classe `HandlerMapping`

Cette classe permet de déterminer quel contrôleur doit gérer les requêtes sur une URL précise : elle fait la correspondance (le « mapping ») entre une URL et une méthode d'un contrôleur.

Elle est en fait invisible de l'utilisateur, on se contente de la configurer.

Spring MVC

Association URL / méthode de contrôleur

Association URL / méthode de contrôleur

On peut la configurer au travers :

- des balises `<servlet-mapping>` du fichier `web.xml` situé dans le répertoire `WEB-INF` de l'application web
- d'annotations comme `@RequestMapping`, `@GetMapping`, `@PostMapping`, etc. Ces annotations sont portées par les méthodes des contrôleurs, qui sont les objets traitant les requêtes HTTP, ou par les contrôleurs eux-mêmes.

Association URL / méthode de contrôleur

Une annotation `@RequestMapping` placée sur un contrôleur permet de **partager** des informations de mapping par toutes les méthodes de ce contrôleur.

Il s'agit d'une annotation générique, permettant de mapper une URL sur une méthode d'un contrôleur ou sur toutes les méthodes de ce contrôleur.

Exemple page suivante.

Association URL / méthode de contrôleur

```
@Controller
@RequestMapping("/livres")
class LivreController {
    @RequestMapping("/{isbn}")
    public Person getLivre(@PathVariable Long isbn) {
        // ...
    }
}
```

Association URL / méthode de contrôleur

Dans l'exemple précédent, les requêtes traitées par la méthode `getLivre()` devront porter sur une URL de la forme `/livres/{isbn}`, pas `{isbn}`.

La méthode suivante traiterait les requêtes portant sur l'URL `/livres`.

```
public void addLivre(...) {  
    // ...  
}
```

Association URL / méthode de contrôleur

`@RequestMapping` est une annotation générique, permettant de mapper une URL sur une méthode d'un contrôleur.

On peut ainsi préciser différents paramètres de mapping :

- L'URL associée
- La ou les méthode(s) HTTP
- Les headers HTTP

Association URL / méthode de contrôleur

- Le ou les paramètre(s) de la requête
- Les types de média (media types) consommés
- Le type de média produit

Exemples :

```
@RequestMapping(value="/un/chemin/")
```

→ toutes les requêtes sur l'URL `/un/chemin`, quelle que soit la méthode HTTP utilisée, les paramètres, le type de donnée fourni et produit...

Association URL / méthode de contrôleur

Exemples :

```
@RequestMapping (value="/un/chemin/",  
method=RequestMethod.GET)
```

→ toutes les requêtes sur l'URL /un/chemin, si la méthode HTTP utilisée est GET, quels que soient les paramètres, le type de donnée fourni et produit...

Association URL / méthode de contrôleur

Exemples :

```
@RequestMapping (value="/un/chemin/",  
method={RequestMethod.POST,  
RequestMethod.PUT})
```

→ toutes les requêtes sur l'URL /un/chemin, si la méthode HTTP utilisée est POST ou PUT, quels que soient les paramètres, le type de donnée fourni et produit...

Association URL / méthode de contrôleur

Exemples :

```
@RequestMapping (value="/un/chemin/",  
headers="cle=valeur")
```

→ toutes les requêtes sur l'URL /un/chemin, quelle que soit la méthode HTTP utilisée, quels que soient les paramètres, le type de donnée fourni et produit, à condition que l'association `cle=valeur` soit présente dans les headers HTTP.

Association URL / méthode de contrôleur

Exemples :

```
@RequestMapping (value="/un/chemin/",  
produces="application/json")
```

→ toutes les requêtes sur l'URL /un/chemin, quelle que soit la méthode HTTP utilisée, quels que soient les paramètres ou le type de donnée fourni, à condition que le type de donnée produit soit du JSON.

Association URL / méthode de contrôleur

Exemples :

```
@RequestMapping (value="/un/chemin/",  
headers="Accept=application/json")
```

→ une autre manière de mapper les requêtes du même type que celles supportées par l'annotation précédente.

Association URL / méthode de contrôleur

Exemples :

```
@RequestMapping (value="/un/chemin/",  
consumes="text/xml")
```

→ toutes les requêtes sur l'URL /un/chemin, quelle que soit la méthode HTTP utilisée, quels que soient les paramètres ou le type de donnée produit, à condition que le type de donnée fourni soit de l'XML.

Association URL / méthode de contrôleur

Les parties variables d'une URL peuvent être attachées à des paramètres de la méthode concernée au travers de l'annotation `@PathVariable`.

```
@RequestMapping(value="/employees/{id}")  
public Employee getEmployeeById(  
    @PathVariable("id") long id) {  
    ...  
}
```

Association URL / méthode de contrôleur

L'identifiant de la partie variable peut être négligé si non nécessaire.

```
@RequestMapping(value="/employees/{section}/{id}")
```

```
public Employee getEmployeeById(  
    @PathVariable String sectionName,  
    @PathVariable long employeeId) {
```

Association URL / méthode de contrôleur

Les paramètres d'une requête HTTP peuvent être automatiquement mappés sur des paramètres de la méthode concernée au travers de l'annotation

`@RequestParam`.

```
@RequestMapping(value="/employees")
```

```
public Employees getEmployees(
```

```
    @RequestParam(name="section") String section,
```

```
    Model model) {
```

```
    ...
```

Association URL / méthode de contrôleur

Il est possible de préciser si le paramètre de la requête HTTP est obligatoire ou pas.

Si aucune valeur n'est fournie, alors une valeur par défaut pour ce paramètre peut être précisée.

```
@RequestMapping(value="/employees")  
  
public Employees getEmployees(  
    @RequestParam(name="section", required=false,  
    defaultValue="finance") String section, Model model) {  
  
    ...  
}
```

Association URL / méthode de contrôleur

Comme d'habitude, Spring fournit de nombreuses autres possibilités d'associer une URL à une ou plusieurs méthodes de contrôleur ou paramètres de méthode de contrôleur...

Spring MVC

Les contrôleurs

Les contrôleurs

Les contrôleurs sont en charge de réaliser les fonctions effectives.

Ils sont activés par le contrôleur central, qui leur transmet les informations utiles sur cette requête (paramètres, variables de path, données attachées...), ou la requête HTTP elle-même.

Comme vu à l'instant, ces informations peuvent être indiquées au moyen d'annotations.

Les contrôleurs

Il s'agit de **beans**, qui peuvent (doivent) être construits et configurés par le container IoC de Spring.

La classe qui implémente un contrôleur peut porter l'annotation `@Controller` pour indiquer à l'IoC container qu'il s'agit d'un bean particulier : un contrôleur MVC.

Cette classe comporte les méthodes traitant les requêtes HTTP.

Comme on l'a vu, l'association entre l'URL des requêtes et les méthodes du contrôleur est faite par annotation.

Les contrôleurs

Les contrôleurs réalisent leur fonction et en indiquent le résultat sous forme d'un **modèle** (interface `org.springframework.ui.Model`).

Ils doivent également indiquer au contrôleur central quelle **vue** permet d'afficher le modèle retourné.

Les contrôleurs

Le modèle peut être indiqué de différentes manières.

En général, le modèle est représenté par une map (alias dictionnaire) `java.util.Map`, c'est-à-dire un ensemble de couples clé/valeur.

La clé est le nom d'un attribut, et la valeur est la valeur de cet attribut. Le type de cette valeur est quelconque.

Le modèle est donc généralement représenté par un ensemble d'attributs.

Les contrôleurs

Exemple.

Supposons qu'un contrôleur a recherché dans une base de données des informations sur une voiture donnée.

Les informations que l'on veut afficher seraient le numéro d'identification unique du véhicule, sa marque, son modèle, son année de fabrication, et son type de carburant.

Le modèle correspondant à cet exemple pourrait être le suivant.

Les contrôleurs

```
{  
  "id" : "AL-703-QB",  
  "marque" : "Ford",  
  "modele" : "Focus",  
  "annee" : 2008,  
  "carburant" : "essence"  
}
```

Les contrôleurs

De tels attributs peuvent être ajoutés à un objet `Model` grâce aux méthodes suivantes :

```
addAllAttributes (Map<String, ?> attributes)
```

```
addAttribute (Object attributeValue)
```

```
addAttribute (String attributeName, Object  
attributeValue)
```

```
mergeAttributes (Map<String, ?> attributes)
```

Les contrôleurs

D'autres méthodes de cet objet `Model` permettent de le manipuler :

```
Map<String, Object> asMap()
```

```
boolean containsAttribute(String attributeName)
```

```
Object getAttribute(String attributeName)
```

La vue en charge d'afficher le modèle peut ainsi accéder aux informations du modèle pour générer la représentation correspondante.

Les contrôleurs

Comme toujours, Spring permet de gérer l'interface entre le contrôleur central et les contrôleurs de différentes manières. Spring supporte **33** types d'arguments différents pour les méthodes de contrôleur...

Ils peuvent lui retourner une `String` contenant l'identifiant de la vue et mettre à jour le modèle sous forme d'une instance de `Model`.

```
@GetMapping
```

```
public String handle(Model model) { ... }
```

Les contrôleurs

Ils peuvent aussi retourner l'identifiant de vue sous forme d'une `String` et le modèle sous forme d'une `Map<String, ?>`.

```
@GetMapping
```

```
public String getLivre(@PathParam int isbn,  
Map<String, ?> result) { ... }
```

Les contrôleurs

Une possibilité équivalente : retourner l'identifiant de vue sous forme d'une `String` et le modèle sous forme d'une `ModelMap`.

Une `ModelMap` est une `LinkedHashMap<String, Object>`.

```
@GetMapping
```

```
public String getLivre(@PathParam int isbn,  
ModelMap result) { ... }
```

Les contrôleurs

Les contrôleurs peuvent également retourner au contrôleur central un objet `ModelAndView` décrivant le nouveau modèle à afficher et le moyen d'afficher ce modèle.

L'idée est de pouvoir grouper dans une seule valeur de retour l'identifiant de la vue et le modèle à afficher.

Car Java n'est pas Python...

Les contrôleurs

```
import org.springframework.web.servlet.ModelAndView;  
  
...  
  
@GetMapping(value = "/display/{id}")  
  
public ModelAndView afficher(@PathVariable String  
id) {  
  
    ...  
  
}
```

Les contrôleurs

Exemple de mise en œuvre d'un ModelAndView.

```
@GetMapping(value = "/hello")  
  
public ModelAndView salut(@RequestParam(name="nom",  
required=false, defaultValue="world") String nom) {  
    ModalAndView resultat = new ModalAndView("salut");  
    resultat.addObject("nom", nom);  
    return resultat;  
}
```

Ajout d'un attribut
au modèle

Identifiant
de la vue

Les contrôleurs

On a vu que le contrôleur doit retourner non seulement un modèle mais aussi ce qui permet de l'afficher.

Ceci peut être fait en retournant directement un objet implémentant l'interface

`org.springframework.web.View`.

Les contrôleurs

Une implémentation de l'interface

`org.springframework.web.View` a pour rôle de générer l'affichage (HTML en général) qui sera pris en compte par le navigateur. Cet affichage expose bien sûr les informations contenues dans le modèle. La méthode correspondante de l'interface est `render()`.

```
void render(Map<String, ?> model,  
            HttpServletRequest request,  
            HttpServletResponse response)
```


Spring MVC

La résolution de vues

La résolution de vue

Plus classiquement, et de manière plus souple et évolutive, la vue à afficher peut être indiquée par un contrôleur en retournant un **identifiant** de cette vue.

Cet identifiant de vue permettra de découvrir la vue correspondante lors du processus de **résolution de vue**.

La vue correspondante (i.e. une implémentation de l'interface View) sera retournée par le **résolveur de vues**, un **bean** implémentant l'interface

`org.springframework.web.servlet.ViewResolver`.

La résolution de vue

On voit ainsi que Spring MVC permet de découpler l'application web de la technologie de génération des pages HTML.

En effet, un contrôleur ne génère pas la page HTML qu'il renvoie, il retourne plutôt un identifiant correspondant à une **vue** qui, elle, générera la page HTML, typiquement par analyse du modèle dans un template HTML.

La **résolution de vue** est le processus par lequel cet identifiant permet de découvrir la vue correspondante.

La résolution de vue

Spring propose différentes classes pour réaliser la résolution de vue :

- `UrlBasedViewResolver`
- `InternalResourceViewResolver`
- `XmlViewResolver`
- `ResourceBundleViewResolver`
- ...

La résolution de vue

La classe `UrlBasedViewResolver` permet de trouver la vue correspondant à un identifiant de vue de manière très simple, par recherche d'une ressource (fichier HTML ou JSP par exemple) dont le nom correspond directement à l'identifiant de vue, au suffixe près.

Le suffixe de l'URL ainsi que le préfixe (localisation de ces ressources) peuvent être précisés par configuration de ce résolveur.

La résolution de vue

Exemple en XML.

```
<bean id="viewResolver"  
class="org.springframework.web.servlet.view.UrlBasedViewR  
esolver">  
  
    <property name="prefix" value="/WEB-INF/jsp/" />  
  
    <property name="suffix" value=".jsp" />  
  
    <property name="viewNames" value="*jsp" />  
  
</bean>
```

La résolution de vue

La classe `InternalResourceViewResolver` est une dérivation de la classe précédente.

Elle permet principalement d'associer un fichier JSP ou JSTL à un identifiant de vue.

Comme pour le résolveur précédent, il est possible de préciser le préfixe et le suffixe à utiliser pour trouver les ressources. Et comme toujours avec Spring, on peut spécifier cela par fichier XML, par programme ou par ressource.

La résolution de vue

Exemple de configuration d'un bean retournant un `InternalResourceViewResolver` et configuré par programme.

@Bean

```
public ViewResolver configureViewResolver() {  
    InternalResourceViewResolver viewResolver = new  
InternalResourceViewResolver();  
    viewResolver.setPrefix("/templates/views/");  
    viewResolver.setSuffix(".jsp");  
  
    return viewResolver;  
}
```


La résolution de vue

Que représente le code précédent et où doit-il être placé ?



La résolution de vue

Rappel sur l'injection de dépendances.

Il s'agit de code Java de configuration de Spring (« Java-based Spring configuration »).

Il peut être placé dans tout package scanné par Spring, que ce soit au travers d'une directive `<component-scan>` d'un fichier XML de configuration ou si l'auto-configuration a été activée par les annotations `@Configuration`, `@Bean` et `@ComponentScan`.

La résolution de vue

Une autre possibilité pour configurer un `InternalResourceViewResolver` est d'utiliser un fichier de ressources.

Exemple de contenu d'un fichier `.properties`.

```
spring.mvc.view.prefix=/WEB-INF/jsp/
```

```
spring.mvc.view.suffix=.jsp
```

La résolution de vue

La recherche de vues peut aussi être faite en recherchant un bean correspondant à l'identifiant de vue parmi les beans de type `View` déclarés dans un fichier XML.

La classe `XmlViewResolver` est une classe supportant ce mécanisme de configuration.

Par défaut, le fichier XML de définition des beans de type `View` est le fichier `views.xml` placé dans le répertoire `/WEB-INF/`.

La résolution de vue

Exemple.

Supposons qu'un contrôleur retourne comme identifiant de vue la chaîne "Bonjour".

Si le résolveur de vue est un `XmlViewResolver`, il cherchera dans la liste des beans de type `View` qu'il connaît une vue dont le nom serait `Bonjour`.

On pourrait préciser ainsi dans le fichier XML de définition des beans de l'application que c'est le fichier `/WEB-INF/mes_vues.xml` qui définit ces vues.

La résolution de vue

Fichier XML de définition des beans de l'application.

...

```
<bean  
class="org.springframework.web.servlet.view.XmlViewResolver">  
    <property name="location">  
        <value>/WEB-INF/mes_vues.xml</value>  
    </property>  
</bean>
```

La résolution de vue

Ensuite, le fichier `mes_vues.xml` de définition des vues de l'application pourrait définir cette vue ainsi.

...

```
<bean id="Bonjour"  
class="org.springframework.web.servlet.view.JstlView">  
    <property name="url" value="/WEB-INF/pages/Salut.jsp" />  
</bean>
```

La résolution de vue

On voit qu'il n'y a plus un mapping immédiat entre l'identifiant de la vue et l'URL de la ressource : le fichier de définition des beans de type View permet de réaliser un mapping arbitraire entre ces deux informations.

"Bonjour" → /WEB-INF/pages/Salut.jsp.

Alors qu'avec un `UrlBasedViewResolver`, le mapping est direct :

"Bonjour" → /WEB-INF/pages/Bonjour.jsp.

La résolution de vue

Un autre résolveur de vue permet de spécifier u mapping arbitraire : `ResourceBundleViewResolver`.

Un tel résolveur recherche le mapping au travers d'un fichier de ressources (resource bundle), c'est-à-dire typiquement un fichier de propriétés Java.

Le nom du fichier de propriétés est `views` par défaut. Il peut être défini au moyen de la propriété `basename` du résolveur de vue.

La résolution de vue

L'identifiant d'une vue et son type doivent être définis ainsi dans le fichier de ressources.

```
<id de la vue>.class=<classe de la vue>
```

```
<id de la vue>.url=<url de la vue>
```

Par exemple :

```
Accueil.class=org.springframework.web.servlet.view.JstlView
```

```
Accueil.url=/WEB_INF/pages/main.jsp
```

La résolution de vue

Exemple : fichier de définition des beans de l'application.

```
<beans ...>

    <bean
      class="org.springframework.web.servlet.view.ResourceBu
ndleViewResolver">
        <property name="basename" value="vues" />
    </bean>
</beans>
```

La résolution de vue

Dans le cas où un contrôleur retournerait `Login` comme identifiant de vue, si on voulait lui associer la page JSP `identification.jsp`, le fichier `vues.properties` devrait contenir ce qui suit.

```
Login.class=org.springframework.web.servlet.view.InternalResourceView
```

```
Login.url=/WEB_INF/pages/identification.jsp
```

La résolution de vue

Autres points à savoir sur la résolution de vues :

- Plusieurs résolveurs de vue peuvent être **chainés**, l'ordre d'utilisation des résolveurs étant fixé par configuration au moyen de la propriété `order` sur ces différents résolveurs
- Les résolveurs peuvent supporter l'**internationalisation**, comme quasiment tous les composants de Spring
- Etc...

Spring MVC

La génération de vue

La génération de vue

Spring permet ainsi d'intégrer différentes technologies de génération de vue :

- JSP (Java Server Pages) et JSTL (JSP standard tag library)
- Thymeleaf (un moteur de templating)
- FreeMarker (un autre moteur de templating, d'origine Apache, capable de générer du HTML mais aussi d'autres formats)
- Groovy Markup (encore un autre moteur de templating)

La génération de vue

Tout moteur de templating compatible de la **JSR-223** (langages de script pour Java) peut en fait être intégré avec Spring.

Spring peut aussi (faire) générer des vues avec un format autre que l'HTML :

- De l'XML
- Du PDF
- De l'Excel
- Du JSON

La génération de vue

Spring MVC peut aussi supporter un **moteur XSLT** comme moteur de génération de vues.

XSLT (Extensible Stylesheet Language Transformation) est un langage permettant de convertir de l'XML en à peu près n'importe quoi.

C'est donc adapté au cas où le modèle retourné par un contrôleur est représenté en XML.

La génération de vue

Utilisation de Thymeleaf

Il faut ajouter les fichiers JAR suivants comme dépendances du projet :

- `thymeleaf-<version>.jar`
- `thymeleaf-spring5-<version>.jar.`

Ceci peut être fait simplement au moyen de la directive Maven suivante.

La génération de vue

```
<dependency>
```

```
  <groupId>org.springframework.boot</groupId>
```

```
  <artifactId>spring-boot-starter-thymeleaf</artifactId>
```

```
</dependency>
```

La génération de vue

Par défaut, Thymeleaf recherche des fichiers source suffixés `.html` et placés dans le répertoire `templates` sous `src/main/resources` (cas d'une organisation du projet compatible avec Maven) ou `static` pour les pages statiques.

Cette configuration par défaut peut être modifiée, par exemple en précisant le préfixe (localisation) et le suffixe des fichiers source.

La génération de vue

Utilisation de pages JSP

Il faut ajouter les dépendances suivantes :

- Jasper, le moteur de JSP d'Apache
 - `tomcat-embed-jasper-<version>.jar`
- La librairie standard de tags JSTL (extension Java)
 - `jstl-<version>.jar`.

Ceci peut être fait simplement au moyen des directives Maven suivantes.

La génération de vue

```
<dependency>
```

```
  <groupId>org.apache.tomcat.embed</groupId>
```

```
  <artifactId>tomcat-embed-jasper</artifactId>
```

```
</dependency>
```

```
<dependency>
```

```
  <groupId>javax.servlet</groupId>
```

```
  <artifactId>jstl</artifactId>
```

```
</dependency>
```

La génération de vue

Les fichiers source JSP sont recherchés sous le répertoire-racine de l'application web, c'est-à-dire `src/main/webapp` dans le cas d'une organisation compatible de Maven.

Comme vu précédemment, la localisation des pages JSP sous ce répertoire-racine et leur suffixe peut être précisé au moyen de directives de configuration dans un fichier XML ou un fichier de propriétés, ou bien en code Java.

La génération de vue

Utilisation de FreeMarker

Il faut ajouter la dépendance suivante :

- FreeMarker JAR → `freemarker-<version>.jar`

Ceci peut être fait simplement au moyen de la directive Maven suivante.

La génération de vue

```
<dependency>
```

```
  <groupId>org.springframework.boot</groupId>
```

```
  <artifactId>spring-boot-starter-freemarker</artifactId>
```

```
</dependency>
```

La génération de vue

Les fichiers source FreeMarker sont suffixés `.ftl` et placés dans le répertoire `templates` sous `src/main/resources` (cas d'une organisation du projet compatible avec Maven) ou `static` pour les pages statiques.

Cette configuration par défaut peut être modifiée en Java en créant une instance de la classe `freemarker.template.Configuration` et en activant diverses méthodes de cette instance.

La génération de vue

Exemples de méthodes de configuration :

- `setDirectoryForTemplateLoading()`
- `setDefaultEncoding()`
- `setTemplateExceptionHandler()`
- **etc.**

Spring MVC

Mise en pratique :

- Mise en oeuvre de Spring Initializr et Spring Boot
- Exercice 01 : une application Spring MVC basique
- Exercice 02 : une application Spring MVC dynamique
- Exercice 03 : une application Spring MVC riche



Spring MVC

Autres points

Alternative à Spring Boot

On peut (faire) configurer une application web Spring MVC au moyen de Spring Boot Starter.

On peut aussi la configurer de manière plus classique (et économe en dépendances...) avec le module Spring MVC `spring-webmvc`.

```
@Configuration
```

```
@EnableWebMvc
```

```
@ComponentScan(basePackages = "mes.controleurs")
```

```
public class WebConfig { ... }
```

Alternative à Spring Boot

Ceci évite, là aussi, tout recours à un fichier XML de définition de l'application web.

La dépendance Maven à indiquer est alors la suivante.

```
<dependency>  
    <groupId>org.springframework</groupId>  
    <artifactId>spring-webmvc</artifactId>  
</dependency>
```

Gestion des erreurs

Si le résolveur de vue ne peut trouver la vue associée à un identifiant retourné par un contrôleur, alors Spring Boot recherche une page mappée sur l'identifiant de vue `error` (**genre** `error.html`, `error.jsp` ou `error.ftl` ou tout autre nom déterminé par le résolveur de vue).

Si cette page ne peut être trouvée, Spring Boot retourne une page HTML appelée « whitelabel page ».

Gestion des erreurs

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Sun Nov 24 16:31:31 CET 2019

There was an unexpected error (type=Not Found, status=404).

No message available

Gestion des erreurs

Il est possible d'indiquer une page d'erreur commune, qui sera affichée en cas de problème lors du traitement d'une requête.

Pour cela, il suffit de définir une page mappée sur l'identifiant de vue `error`.

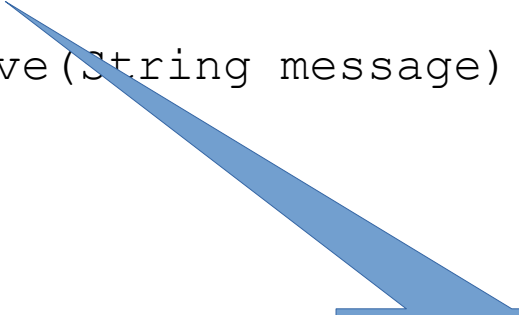
Gestion des erreurs

Que se passe t'il si une exception « user-defined » (définie par le développeur) ou de type `RuntimeException` est levée lors du traitement d'une requête dans un contrôleur ?

Comme pour une exception générée par Spring, une page mappée sur l'identifiant de vue `error` sera recherchée. Si elle n'est pas trouvée, une page whitelabel sera envoyée, avec le texte de l'exception insérée à la fin de la page.

Gestion des erreurs

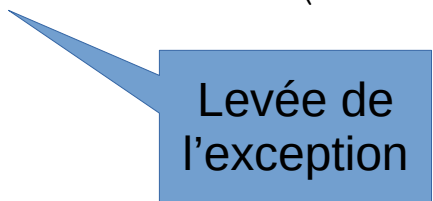
```
public class Bibliotheque {  
    public class LivreNonTrouve extends RuntimeException {  
        public LivreNonTrouve(String message) {  
            super(message);  
        }  
    }  
    ...  
}
```



Exception définie
par le développeur

Gestion des erreurs

```
public Livre trouverLivre(String isbn) throws  
Bibliotheque.LivreNonTrouve {  
  
    if (this.bibliotheque.containsKey(isbn)) {  
  
        return this.bibliotheque.get(isbn);  
  
    }  
  
    throw new Bibliotheque.LivreNonTrouve("Livre avec  
ISBN " + isbn + " non trouve !");  
  
}
```



Levée de l'exception

Gestion des erreurs

```
@GetMapping("/biblio/chercher")

public String chercherLivre(@RequestParam(name = "isbn",
required = true) String isbn, Model modele) throws
LivreNonTrouve {

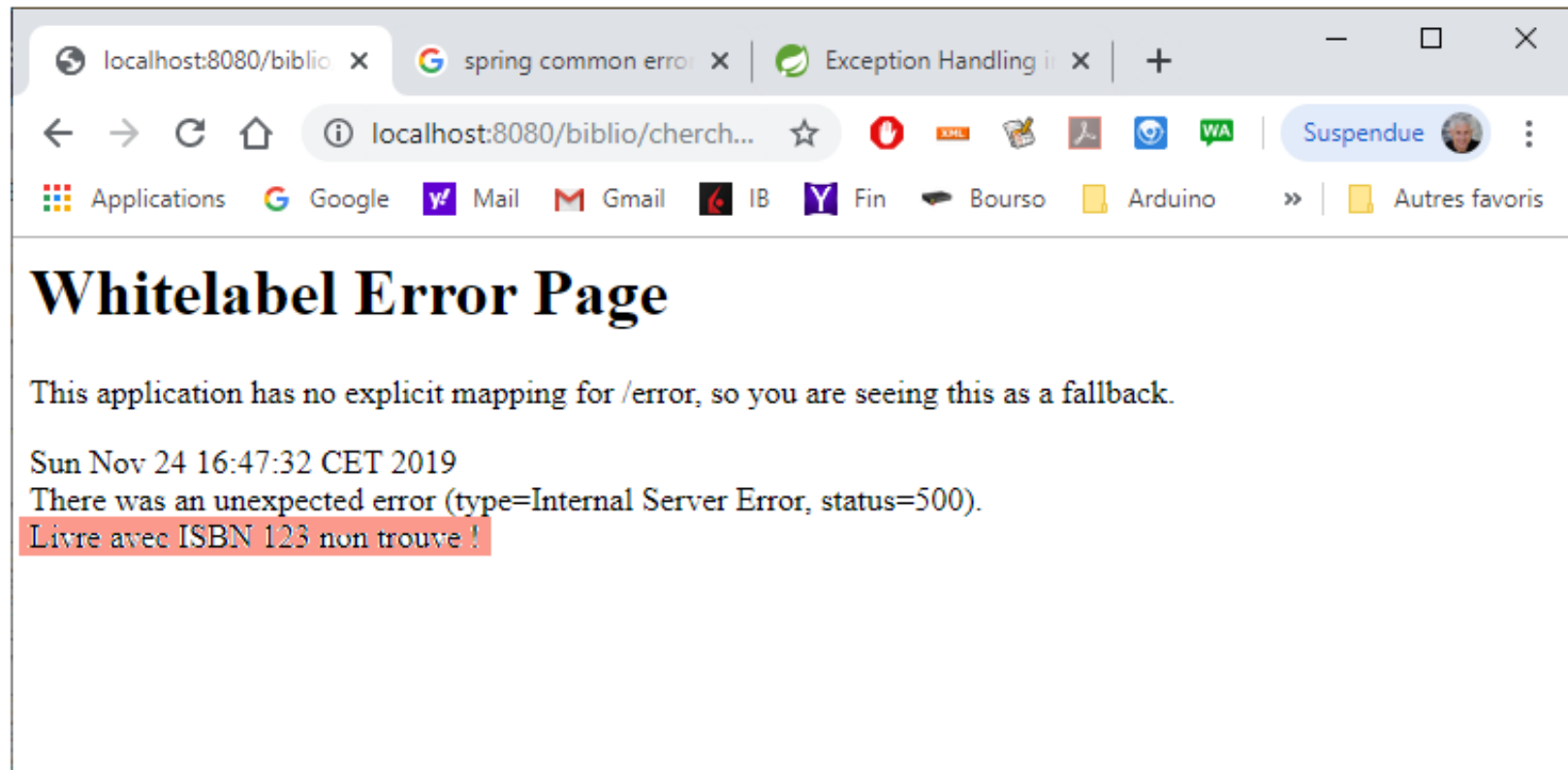
    Livre livre = bibliotheque.trouverLivre(isbn);

    modele.addAttribute("livre", livre);

    return "affichage_livre";

}
```

Gestion des erreurs



Gestion des erreurs

Le code de retour HTTP associé par Spring MVC à une **erreur inattendue** sur le serveur est 500 (`InternalServerError`).

Si Spring ne trouve **pas** de contrôleur gérant une URL, le code de retour HTTP associé à l'erreur sera 404 (`NotFound`).

Gestion des erreurs

Si Spring MVC trouve un contrôleur gérant une URL mais qu'aucune méthode n'est **adaptée**, alors le code de retour HTTP associé à l'erreur sera 405 (Method Not Allowed).

Si Spring détecte un problème de **format de la requête** (paramètres manquants, par exemple), le code de retour HTTP associé à l'erreur sera 400 (Bad Request).

Gestion des erreurs

Il est possible de définir soi-même le code de retour HTTP associé à la levée d'une exception.

Ceci peut être fait (parmi d'autres solutions) en annotant l'exception en question avec `@ResponseStatus`.

L'annotation porte 2 attributs :

- `value` (alias `code`), le code de retour HTTP
- `reason`, un texte associé.

Gestion des erreurs

Exemple.

```
@ResponseStatus(code=HttpStatus.GATEWAY_TIMEOUT,  
reason"Pasque !")  
  
public class ReseauTropLent extends Exception {  
    public ReseauTropLent(String message) {  
        super(message);  
    }  
}
```

Gestion des erreurs

Attention ! HTTP est un protocole **applicatif**, pas un protocole de transport, contrairement à ce que son nom laisse penser.

Autrement dit, un code de retour HTTP a un sens **précis**, qui est **interprété** par les applications (client HTTP, serveur HTTP) mais aussi les équipements réseau (reverse proxy, routeurs, etc.) ⇒ voir le style architectural **REST**.

Gestion des erreurs

La vue d'erreur commune `error` est une vue comme les autres, on peut donc lui associer un modèle contenant toute information utile pour décrire l'erreur : le code de retour HTTP, le texte associé à une exception, un objet Java particulier, etc.

Gestion des erreurs

Par défaut, Spring crée une association entre l'URL `/error` et un contrôleur par défaut, le `BasicErrorController`.

Ce contrôleur retourne `error` comme identifiant de vue ainsi qu'un modèle contenant des informations sur l'erreur (texte de l'exception, code de retour HTTP).

Si le résolveur de vue ne trouve aucune page correspondant à cet identifiant, une page whitelabel sera générée.

Gestion des erreurs

Il est possible de désactiver la page d'erreur whitelabel par configuration de Spring Boot.

Le plus simple consiste à placer la chaine suivante dans le fichier `application.properties`, sous `src/main/resources` :

```
server.error.whitelabel.enabled=false
```

Dans ce cas, c'est le container de servlets qui affichera un message d'erreur.

Gestion des erreurs

Il est aussi possible de remplacer le `BasicErrorController` par un contrôleur réalisé par nos soins.

Il faut pour cela réaliser un contrôleur implémentant l'interface `ErrorController`.

Cette interface contient une seule méthode : `getErrorPath()`, qui indique la partie « path » de l'URL vers la page d'erreur (`/error` par défaut).

Gestion des erreurs

Cette méthode est utilisée par Spring Boot pour savoir à quelle URL envoyer les erreurs apparaissant lors du traitement d'une requête, en amont (lors de la recherche d'une méthode pour traiter la requête HTTP) ou en aval (lors de la résolution de la vue ou de sa génération).

Dans ce contrôleur, on peut ainsi mapper cette URL d'erreur vers une méthode de ce contrôleur qui traitera les erreurs. On réalise donc ainsi la centralisation des erreurs.

Gestion des erreurs

Exemple

@Controller

```
public class ControlleurDErreur implements ErrorController {  
    @RequestMapping("/error")  
    public ModelAndView gere(HttpServletRequest request) {  
        Integer codeHttp = (Integer)  
            request.getAttribute("javax.servlet.error.status_code");  
        Exception exception =  
            request.getAttribute("javax.servlet.error.exception");
```

Gestion des erreurs

```
ModelAndView modele = new ModelAndView("error");  
modele.addObject("codeHttp", codeHttp);  
modele.addObject("exception", exception);  
return modele;
```

```
}
```

@Override

```
public String getErrorPath() {  
    return "/error" ;  
}
```

```
}
```

Gestion des erreurs

Il suffit juste ensuite de définir un template, par exemple `error.jsp` (JSP), `error.html` (Thymeleaf) ou `error.ftl` (FreeMarker).

Ce template extraiera les informations placées dans le modèle pour générer la page d'erreur à afficher.

NB. Le code de retour HTTP est accessible dans un contrôleur au moyen de la méthode

`getAttribute("javax.servlet.error.status_code")` ; sur la requête HTTP (`HttpServletRequest`).

Spring MVC

Mise en pratique :

- Exercice 04 : une application Spring MVC avec gestion des erreurs



Validation des formulaires

Spring fournit aussi un cadre pour simplifier la validation des formulaires (projet Spring Validation).