Spring 3.X

Spring pour le Web

plan

- Configuration de base
- Récupérer un Bean
- Frameworks MVC
- Rappels MVC/MVC2
- Etude Comparative entre les différents frameworks MVC

Configuration de base

- La configuration du conteneur léger est effectuée dans le web.xml
  - Le contexte de l'application est pris en charge via un listener (servlet 2.4 et +)

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

  - En servlet < 2.4, il faudra remplacer listener par la servlet ContextLoaderServlet

```
<servlet>
        <servlet-name>context</servlet-name>
        <servlet-class>
        org.springframework.web.context.ContextLoaderServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
</servlet>
```

Configuration de base

- Les fichiers de configuration Spring
  - Par défaut, le fichier de configuration Spring est alors unique. Il s'agit de :

```
/WEB-INF/applicationContext.xml
```

  - Si l'on souhaite utiliser un fichier différent ou plusieurs fichiers, on spécifie dans une balise `<context-param>` du web.xml, le paramètre :

```
<context-param>
          <param-name>contextConfigLocation</param-name>
          <param-value>
               /WEB-INF/conf/spring/config.xml,classpath:test.xml
          </param-value>
</context-param>
```

- Remarquez que dans l'exemple ci-dessus, la ressource test.xml est recherchée dans le classpath de l'application.

Récupérer un Bean

- Le principe reste le même que pour une application non-web, mais cette fois nous obtenons une instance de `WebApplicationContext`
  - Dans une servlet (ou une JSP), nous passerons par une classe utilitaire `WebApplicationContextUtils`

```
WebApplicationContext
appctx=WebApplicationContextUtils.getWebApplicationContext(getServletContext());
Product p=((Product)appctx.getBean("prod"));
```

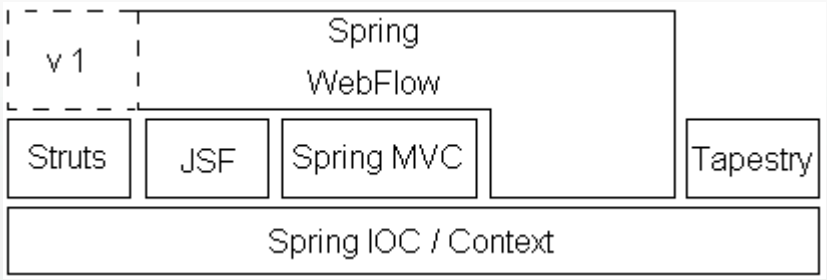
  - Via un framework MVC, ce que nous verrons tout à l'heure.

TP n°1

- Voir document dédié

Frameworks MVC

- Bien-sûr, Spring permet l'intégration de la plupart des frameworks MVC du commerce
  - Struts
  - JSF
  - Tapestry
  - Webwork
- Cependant, Spring propose également son propre framework MVC : Spring MVC
  - Spring propose également une surcouche spécialisée dans la gestion des flots (enchaînements d'écrans) : Spring WebFlow.



Comparatif des solutions

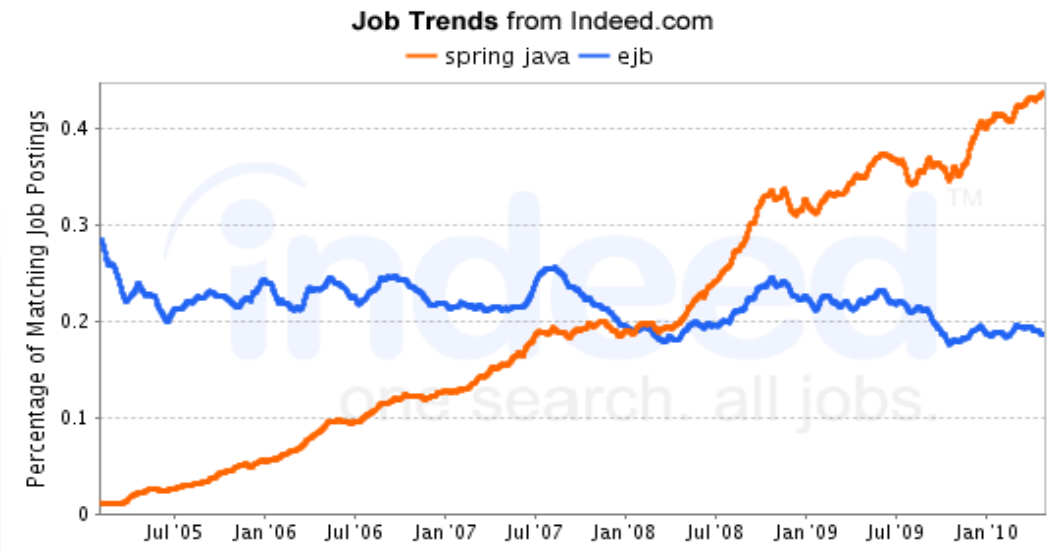
- Alors quel framework choisir ? Les études comparatives entre Spring MVC, JSF, Struts et d'autres framework MVC ne révèlent aucun résultat évident.
- Lorsque l'on cherche à évaluer l'intérêt d'une technologie, on considère différents aspects :
  - La courbe d'apprentissage
  - L'évolutivité
  - La facilité d'intégration
  - La testabilité
  - L'activité de la communauté
  - pourquoi pas le potentiel en terme d'emplois...
  - Etc...

Comparatif des solutions

- Ce qui est indéniable, c'est que Spring MVC présente des particularités intéressantes.
- Spring MVC se veut plus flexible que d'autres framework. Spring MVC propose un découpage fin et clair du mécanisme du contrôleur.
  - Certains diront « trop flexibles », la courbe d'apprentissage est bien supérieure à celle de Struts
- La communauté Spring est très active, ce qui est souvent un critère déterminant dans le choix d'une technologie open-source.
- La documentation est assez complète même si, vu l'ampleur du sujet, on trouve parfois difficilement l'exemple qui correspond exactement à un besoin particulier.

Comparatif des solutions

- Enfin, en terme d'offres d'emploi, on constate que Spring MVC à maintenant rattrapé sont retard sur le standard JSF de Sun (Oracle), principalement car Spring MVC s'intègre tout naturellement à Spring et que Spring à lui déjà dépassé le standard EJB depuis longtemps.



Spring 3.0

Spring MVC

plan

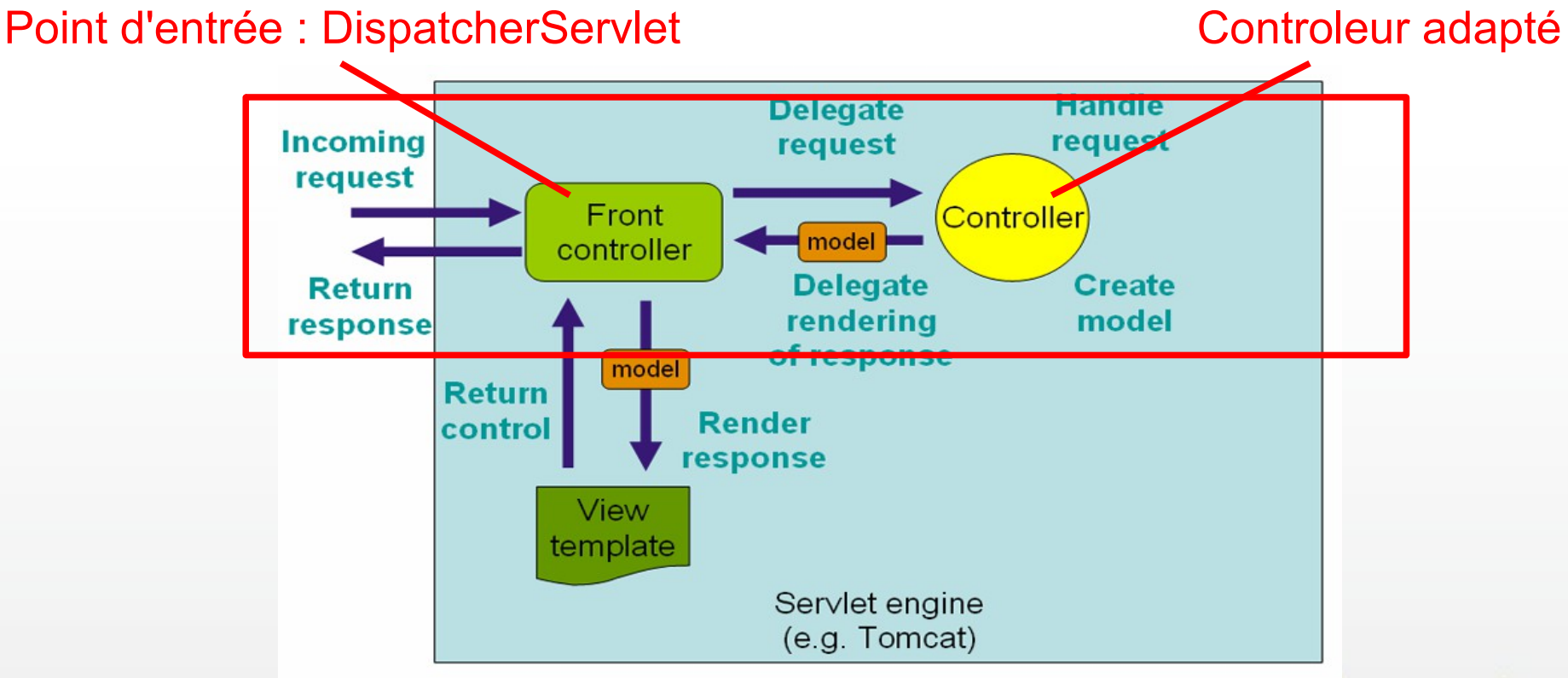
- Le contrôleur
- HandlerMapping
- La vue
- Le modèle
- Compilation et déploiement
- Injection
- Annotations @Controller et @RequestMapping
- Trucs et astuces

plan

- ModelAndView
- Types de retour des méthodes
- Annotation @ModelAttribute
- Annotation @SessionAttributes
- Gestion de formulaires
- Validation de beans – JSR 303
- Scopes pour le Web
- Mixage des scopes
- Controller RestFul / Web Services REST

Le contrôleur

- Le point d'entrée unique de traitement des requêtes au sens MVC2 est assumé dans Spring MVC par la classe `DispatcherServlet` qui délègue le traitement à un contrôleur adapté (équivalent au `ActionServlet` de Struts)



Le contrôleur

- La servlet DispatcherServlet sera déclarée dans le web.xml.

```
<servlet>
     <servlet-name>actions</servlet-name>
     <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
     <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
     <servlet-name>actions</servlet-name>
     <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

- Dès lors, tout ce qui est spécifique à Spring MVC sera indiqué dans un fichier WEB-INF/**actions**-servlet.xml (ou actions provient du servlet-name déclaré.
- Ce fichier n'est pas à référencer dans le web.xml

Spring MVC

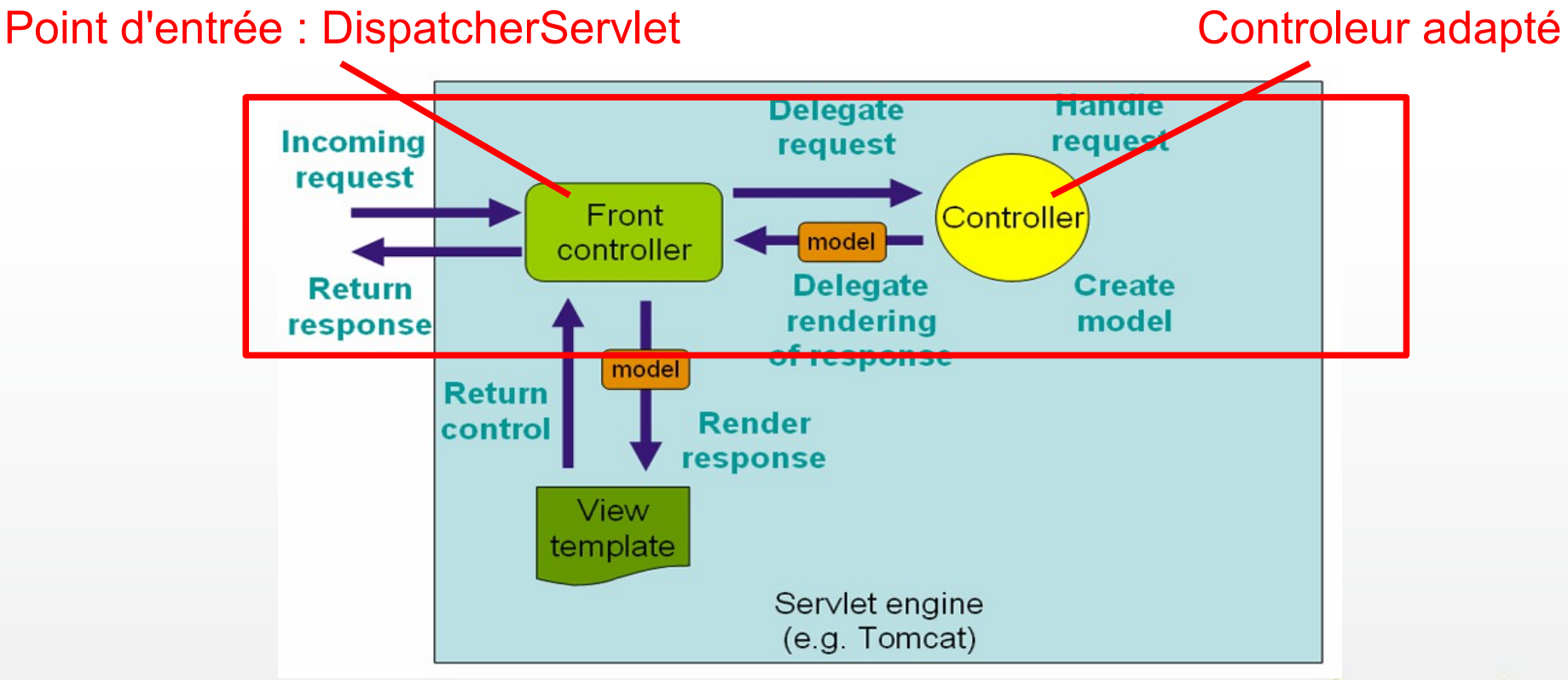
- Si nécessaire la servlet peut être déclarée pour plusieurs types de requêtes et sera alors référencé plusieurs fois dans le web.xml (balise `<servlet>`).
- Le contexte d'application se présente toujours sous la forme d'un `WebApplicationContext`, version étendue de `ApplicationContext` avec des fonctionnalités supplémentaires pour le web
- Or le `WebApplicationContext` est en réalité spécifique au `ServletContext`, il y a donc autant de `WebApplicationContext` que de servlet “Front Controller” déclarées dans le web.xml
- La configuration Spring sera donc répartie Servlet par Servlet par défaut dans le fichier :
  - WEB-INF/[nom-de-la-servlet]-servlet.xml
  - Exemples : WEB-INF/formulaires-servlet.xml

Spring MVC

- Attention, si Spring est capable d'injecter dans les beans issus des fichiers `WEB-INF/[nom-de-la-servlet]-servlet.xml` les beans déclarés par ailleurs dans les fichiers de configuration principaux,l'inverse n'est pas vrai ! Les fichiers principaux ne voient pas ce qui a été déclaré dans les fichiers `WEB-INF/[nom-de-la-servlet]-servlet.xml`.
  - Note : Pour autant, un `<context:component-scan>` ou un `<context:annotation-config>` sera à répéter dans chaque fichier `WEB-INF/[nom-de-la-servlet]-servlet.xml` si nécessaire.

Le contrôleur

- Ce point d'entrée unique (Servlet) délègue le traitement de la requête spécifique à un « Contrôleur adapté ». Il s'agit de la « commande » au sens du Design Pattern « Command ».



Spring MVC

- Un contrôleur au sens Spring MVC implémente l'interface Controller

```
public interface Controller {

    ModelAndView handleRequest(HttpServletRequest request,
HttpServletResponse response) throws Exception;
}
```

Cette classe matérialise donc la “commande“ du design pattern « Command » et équivaut à la classe `Action` de Struts.

- Le résultat du traitement sera mis à disposition dans un objet `ModelAndView` et non pas directement dans un scope
  - Remarque : contrairement aux Struts Actions, la méthode ne renvoie pas l'emplacement lié à la vue mais bien le contenu du modèle ! La méthode peut donc renvoyer null. La redirection vers la vue (`ActionForward` de Struts) est gérée différemment selon le type de contrôleur.

Spring MVC

- L'approche la plus simple consiste à écrire un contrôleur qui hérite `AbstractController`, classe qui implémente `Controller`. Le traitement de la requête se fera alors dans une méthode `handleRequestInternal` :

```
public class HelloController extends AbstractController {
    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest request,
HttpServletResponse response) throws Exception {

        //Avec cet AbstractController, c'est la vue nommée "hello" qui sera appelée
        ModelAndView mav = new ModelAndView("hello");
        //un objet du model est liée à la vue sous le nom "message"
        mav.addObject("message","Hello World !");

        return mav;
    }
}
```

Spring MVC

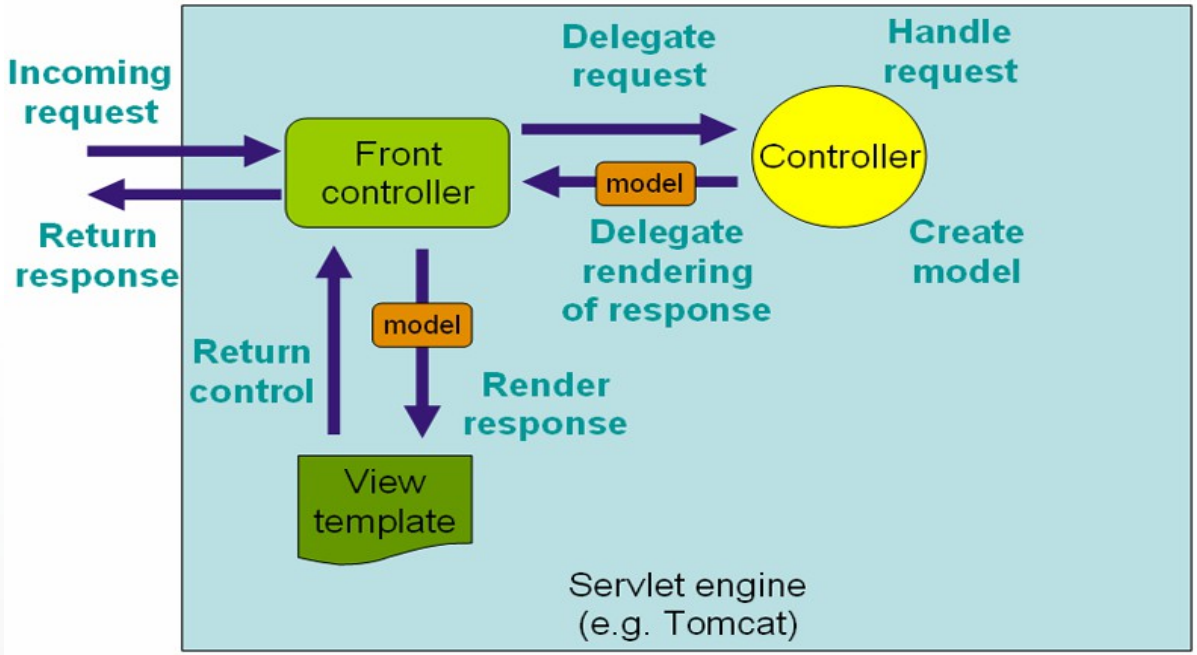
- La valeur passée à la construction du `ModelAndView` représente un identifiant de vue.
- C'est sur la base de cet identifiant que sera déterminé la page à afficher. Nous verrons plus loin comment cette page est déterminée.
- Le contrôleur est un bean comme un autre, son ajout dans le conteneur est donc nécessaire :

```
<bean id="..." name="..." class="formation.HelloController"/>
```

- Nous le verrons plus loin, l'attribut `name` peut avoir une grande importance, l'`id` quant à lui peut tout bonnement être supprimé.

HandlerMapping

- La dernière question qu'il reste à élucider au niveau contrôleur est, comment faire la liaison entre la `DispatcherServlet` et le contrôleur adéquat ?



- C'est là qu'intervient le `HandlerMapping`

HandlerMapping

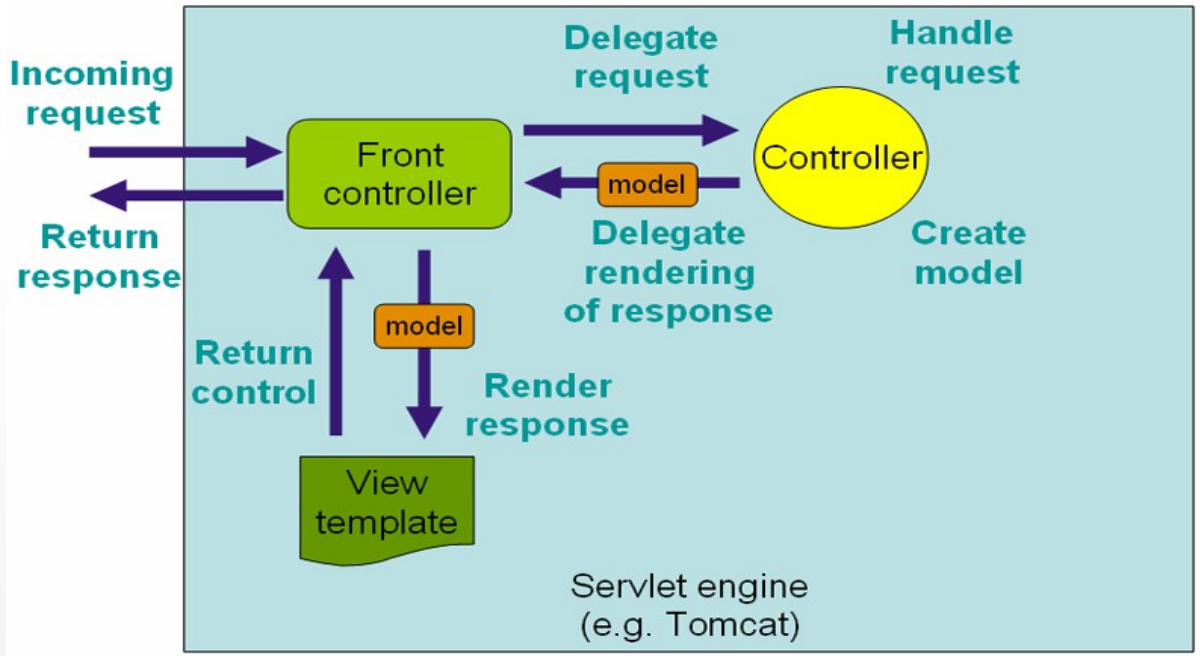
- Il existe plusieurs manières d'établir le lien entre requête et Contrôleur. Prenons un cas simple en considérant la solution suivante, l'usage d'un :
  - `BeanNameUrlHandlerMapping`
- Ce `HandlerMapping` va rechercher le bean ayant un nom exactement identique à l'URI demandée
- La configuration d'un `AbstractController` à base de `BeanNameUrlHandlerMapping` serait la suivante :

```
<bean id="handlerMapping"
class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping" />
<bean name="/test1.do" class="formation.HelloController"/>
```

C'est donc sur la base du nom du Bean que sera déterminé l'url d'accès, ici `/test1.do`.

La vue

- Concrètement, la vue est obtenue grâce à un `ViewResolver`. Il existe là encore un grand nombre de type de `ViewResolver`, examinons le cas simple du `UrlBasedViewResolver`.



La vue

- `UrlBasedViewResolver` redirige vers la page ayant pour nom l'identifiant de la vue préfixé et/ou suffixé
- Exemple : Si l'on veut que la page soit une page JSP / JSTL du même nom que l'identifiant et dans un répertoire pages :

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.UrlBasedViewResolver">
     <property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>
     <property name="prefix" value="/pages/"/>
     <property name="suffix" value=".jsp"/>
</bean>
```

- Le libelle “hello” pour `AbstractController` redirigera donc vers pages/hello.jsp
- Remarquez la propriété `viewClass` qui indique sous quelle forme la vue va être produite, on a choisi ici la vue sous forme JSP/JSTL.
- *Note : Les expressions EL de la JSTL seront bien-sûr disponibles.*

La vue

- D'autres ViewResolver peuvent être plus adaptés à une situation particulière, exemple :
  - `VelocityViewResolver` pour déléguer la production de la vue au framework Velocity

```
<bean id="velocityConfig"
class="org.springframework.web.servlet.view.velocity.VelocityConfigurer">
    <property name="resourceLoaderPath" value="/WEB-INF/velocity/"/>
</bean>

<bean id="viewResolver"
class="org.springframework.web.servlet.view.velocity.VelocityViewResolver">
          <property name="cache" value="true"/>
          <property name="prefix" value=""/>
          <property name="suffix" value=".vm"/>
</bean>
```

- Ce ViewResolver ne se configure pas tout à fait de la même manière que le précédent, la racine de l'emplacement des pages est précisé dans `resourceLoaderPath`.

La vue

- Il est bien évidemment possible de recourir à différents `ViewResolver` en fonction de la demande.
  - Soit en déclarant plusieurs `DispatcherServlet`, et c'est la solution à préférer autant que possible
  - Soit, et cela est beaucoup moins évident, en utilisant un `ResourceBundleViewResolver`. Les différentes associations URL / ViewResolver seront alors décrites dans un resource bundle (fichier de propriétés).

Le Modèle

- Concernant le Modele, Spring reste là très simple en utilisant une Map.
  - `ModelAndView` délègue un `ModelMap` hérité de `java.util.Map`

```
ModelAndView mv=new ModelAndView("hello");
mv.addObject("unObjet", new ActeurImpl("dede","lapoisse"));
return mv;
```

- Toutes ces informations seront ajoutées par Spring MVC en tant que attributs de scope request et donc récupérables dans la vue :

```
<%
    MyObject f=(MyObject)request.getAttribute("unObjet");
    out.println(f.getNom());
 %>
```

Ou bien simplement via une expression EL

```
${unObjet.nom}
```

- Nous le verrons, le passage d'un attribut en scope session sera géré différemment.

Compilation et déploiement

- Au final nous aurons besoin des librairies supplémentaires suivantes :
  - org.springframework.web.servlet.jar
  - jstl.jar
  - standard.jar

TP n°2

- Voir document dédié

TP n°2 Suite [Groupe en avance]

- Voir document dédié

L'injection

- Pour récuperer un bean en provenance du contexte on peut tout à fait utiliser la même méthode que dans une Servlet :

```
protected ModelAndView handleRequestInternal(HttpServletRequest arg0,
                HttpServletResponse arg1) throws Exception {

          WebApplicationContext
context=WebApplicationContextUtils.getWebApplicationContext(getServletContext());
          //ou même plus simplement
          //WebApplicationContext context=getWebApplicationContext();
          ModelAndView mav=new ModelAndView("hello");
          AnyService service=(AnyService)context.getBean("myService");
          mav.addObject(service.getAResult());
          return mav;
     }
```

- Le contrôleur est cependant un bean comme un autre. On pourra y injecter n'importe quel autre bean en provenance des fichiers de configuration ou annotés.

```
     @Resource(name="myService")
     private AnyService service;

     @Override
     protected ModelAndView handleRequestInternal(HttpServletRequest arg0,
                HttpServletResponse arg1) throws Exception {

          ModelAndView mav=new ModelAndView("hello");
          mav.addObject(service.getAResult());
          return mav;
     }
```

TP n°3

- Voir document dédié

@Controller

- Comme nous l'avions déjà introduit, il n'est pas obligatoire de déclarer son contrôleur dans la configuration Spring.
- Nous pouvons demander à Spring d'auto-detecter l'annotation `@Controller` (héritée de `@Component`)
- Encore une fois, cela n'est possible que si l'on précise la balise suivante :

```
<context:component-scan base-package="....controller" />
```

- L'héritage de `AbstractController` n'est plus nécessaire
- Cependant, utiliser l'annotation ne suffira pas.
- En effet, dès lors que l'on ne déclare plus le contrôleur dans la configuration XML, la propriété name n'est de fait plus déclarée et le HandlerMapping qui était basé sur cette propriété devient inopérant.

@RequestMapping

- En réalité,nous aurions pu nous passer de la déclaration du HandlerMapping. Dans ce cas le travail est pris en charge par un HandlerMapping par défaut qui fonctionne à base d'annotations : `DefaultAnnotationHandlerMapping`
  - Ce HandlerMapping attend une (ou plusieurs) annotations `@RequestMapping`
- C'est dans cette balise que l'on indiquera la (ou les) chemins qui permettent d'accéder au contrôleur.

```
@RequestMapping("/hello")
```

- Remarquez l'absence de .do, l'extension devient optionnelle (c'est d'ailleurs mieux ainsi, cette extension devient réellement paramétrable dans le web.xml, ce qui n'était pas le cas dans les exemples précédents.)

@RequestMapping

- L'annotation `@RequestMapping` se place habituellement au niveau de la ou des méthodes qui vont traiter les requêtes.
- Ces méthodes peuvent avoir n'importe quel nom, une gamme de prototype très large (request et response pourront par exemple faire partie des attributs), n'importe quelle visibilité et une gamme de types de retour également très large.
  - Les méthodes retournent cependant habituellement le ModelAndView ou la String identifiant la vue.
- On pourra donc répartir méthode par méthode les différentes fonctionnalités d'une même problématique métier.

```
@RequestMapping("/hello")
private ModelAndView bonjour(HttpServletRequest request, HttpServletResponse response)
throws Exception {
        ModelAndView mav = new ModelAndView("hello");
        mav.addObject("message","Hello World !");
        return mav;
}
@RequestMapping("/bye")
public String aurevoir() {

        return "ciao";
}
```

@RequestMapping

- L'annotation `@RequestMapping` peut également être ajoutée au niveau de la classe (en plus de celles situées au niveau des méthodes). Dans ce cas le chemin indiqué constituera un prefixe au chemin indiqué au niveau de la méthode :

```
@Controller
@RequestMapping("/what")
public class HelloController {


    @RequestMapping("/bye")
    public String aurevoir() {

        return "ciao";
    }
}
```

Le chemin devient ici `/what/bye.do`

TP n°4

- Voir document dédié

ModelAndView

- La classe `ModelAndView` contient à la fois le modèle et la vue.
- Nous l'avons vu, le modèle est exprimé à l'aide d'un `ModelMap` hérité de `java.util.Map`
  - Nous l'avons alimenté avec la méthode `addObject` mais sachez que depuis Java 5, le `ModelMap` complet aurait aussi pu être fourni par le biais de l'interface `Model`
- La vue est elle aussi exprimée à l'aide d'une structure que nous n'avons pas encore étudiée, l'interface `View` et toutes ses implémentations spécifiques au type de vue (JSP, Velocity, Excel etc...) .

Types de retour des méthodes

- Nous avons vu que les méthodes des contrôleurs retournaient habituellement une instance de ModelAndView ou un String identifiant la vue.
- En réalité d'autres types de retour sont possibles :
  - void : la vue sera déterminée sur la base de l'uri qui a permis d'accéder à la méthode (attention : concaténation de tous les @RequestMapping.
  - Model : interface capable de fournir un ModelMap, la vue sera déterminée comme avec void
  - Map : Puisque ModelMap est hérité de Map, idem pour la détermination de la vue
  - View : Interface déléguée de ModelAndView et qui se charge de la vue
  - N'importe quel autre type : L'objet retourné sera alors ajouté au ModelAndView avec pour identifiant le nom de la classe commençant par une minuscule.
  - Plus quelques spécificités...

@ModelAttribute

- Nous venons de voir que l'instanciation d'un `ModelAndView` n'est pas obligatoire.
  - Un retour de type void va pourtant tout de même transmettre un `ModelAndView` qui sera alors vide.
- Et bien ce `ModelAndView` nous allons tout de même pouvoir l'alimenter en objets pour traitement par la vue grâce à l'annotation `@ModelAttribute`
  - Il s'agit d'une annotation que l'on place au niveau d'un attribut (Bean) de la méthode (ou au niveau de la méthode si le bean est déclaré dans le type de retour), bean qui sera instancié et ajouté au `ModelAndView` sous l'identifiant précisé dans l'annotation.

```
@RequestMapping("dovoid")
    private void doVoid(@ModelAttribute("clio") Voiture v) {

        v.setNom("Clio");
        v.setPlaque("AA 252 BB");

    }
```

Autres types de paramètres

- En plus d'une instance de (Http)ServletRequest, d'une instance de (Http)ServletResponse et d'un ensemble de @ModelAttribute, les méthodes acceptent de multiples types d'objets et notamment.
  - Une instance de `Model` (dernier argument impérativement) qui servira à alimenter le `ModelAndView`

```
@RequestMapping("dovoid")
    private void doVoid(@ModelAttribute("clio") Voiture v, Model model) {
        v.setNom("Clio");
        v.setPlaque("AA 252 BB");
        model.addAttribute("message", "another attribute");
    }
```

  - Un ou plusieurs paramètres d'URL annotés alors `@RequestParam` accompagnés éventuellement de `(required=false)`

```
@RequestMapping("dovoid")
    private void doVoid(@ModelAttribute("clio") Voiture v,
@RequestParam("age") int age) {
        v.setNom("Clio");
        v.setPlaque("AA 252 BB");
        v.setAge(age);
    }
```

TP n°5

- Voir document dédié

Gestion des formulaires

- Dans les versions précédentes, les annotations étaient encore peu favorisées. La plupart du temps, la gestion de formulaires passait par l'héritage d'une classe spécifique :
  - SimpleFormController (hérité de CommandController)
- Avec Spring 3.0, cette classe est dépréciée. Nous allons donc travailler uniquement avec les annotations.

Gestion des formulaires

- A la manière d'un `ActionForm` de Struts ou un managed bean JSF, nous allons faire appel à une classe dont les propriétés permettent un mapping avec les champs d'un formulaire HTML. C'est la notion de « backing bean ».
- Avec Spring ces classes sont de simples POJO.
- Spring nomme ces classes « Command ».
  - A ne pas confondre avec les commandes du pattern command dont nous avons déjà parlé (Classes `Controller` Spring).
- La classe est un simple JavaBean ayant des propriétés au nom exact des champs du formulaire.
  - Attention : Utiliser les entités métier existantes comme backing bean est habituellement considéré comme une erreur de conception (Model In View).
  - L'instance qui va se charger de stocker les valeurs des champs HTML sera représenté par un identifiant.

Gestion des formulaires

- Dans les pages jsp, on utilisera la taglib spring-form.tld fournie par Spring dans le jar pour ajouter des formulaires :

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

- Les formulaires seront écrits comme suit :

```
<form:form action="actionUrl" commandName="myNote">
//commandName optionnel → default=command

    <form:input path="personalMessage" />
    <input type="submit" value="valider"/>

</form:form>
```

  - La valeur de `commandName` correspond à l'identifiant du bean de commande (Backing bean).
  - Ici, l'url de soumission du formulaire reste celle qui a permis de conduire au formulaire mais l'attribut action permettrait d'en changer
  - L'attribut `path` correspond à la propriété du Bean.
  - L'ensemble des balises de la taglib est décrit en annexe.

Gestion des formulaires

- Le traitement du formulaire devient un véritable jeu d'enfant

```
@RequestMapping("/creernote")
public String doIt(@ModelAttribute("myNote") Note note){

        //faire quelque chose avec l'objet note.
        System.out.println(note.getPersonalMessage());
        //
        return "note-cree";
}
```

- Un formulaire qui consiste à modifier un « Backing Bean » n'est pas plus compliqué :

```
@RequestMapping("/editionnote")
public String doIt(@ModelAttribute("myNote") Note note){

        //récupérer la note quelque part
        populateFromDatabase(note);
        note.setProprieteSuppl("toto");
        //
        return "form-modification";
}
```

- Les propriétés valorisées viendront compléter les champs de saisie

TP n°6

- Voir document dédié

Validation des beans – JSR 303

- La JSR 303 (javax.validation) a pour objectif de permettre la validation des propriétés des JavaBeans.
- En terme de validation on entend des critères habituels comme :
  - Non null
  - Mini, Maxi
  - Expression régulière etc...
- La JSR 303 ajoute également la possibilité de créer ses propres critères de validation.
- Les critères de validité seront exprimés à l'aide d'annotations spécifiques au niveau des propriétés des Beans.

Validation des beans – JSR 303

- La JSR 303 est une spécification, nous la mettrons en oeuvre en utilisant l'implémentation de référence, celle fournie par Jboss dans le projet Hibernate (indépendant de Hibernate)
- Plusieurs librairies supplémentaires seront nécessaires :
  - validation-api-*.GA.jar
  - hibernate-validator-*.GA.jar
  - slf4j-api-*.jar
  - slf4j-log4j-*.jar
  - log4j*.jar

Validation des beans – JSR 303

- Les annotations qui sont à notre disposition sont les suivantes :
  - @NotNull, @AssertTrue, @AssertFalse, @Size, @Min, @Max, @Pattern, @Past, @Future (et d'autres encore) issues de l'API
  - @NotEmpty, @Length, @Email (et d'autres encore) spécifiques à l'implémentation Hibernate

```
public class UserBackingBean {

    @NotEmpty
    private String nom;

    @Max(value=50)
    private int age;

    @NotEmpty @Pattern(".+@.+\\.[a-z]+")
    private String email;
```

Validation des beans – JSR 303

- En cas d'erreur, un message par défaut pourra être produit, et ceci dans quelques langues courantes.
- On peut cependant produire un message personnalisé en ajoutant l'attribut `message` à nos annotations

```
public class UserBackingBean {

    @NotEmpty(message = "Le champ est obligatoire")
    private String nom;

    ...
```

- On peut également utiliser le « ResourceBundle » d'internationalisation, dans ce cas les propriétés bénéficient d'un nom standardisé :

```
#Appliqué à tous les @NotEmpty
NotEmpty=Le champ est obligatoire
#Appliqué au @NotEmpty de la propriété nom seulement
NotEmpty.nom=Le champ nom est obligatoire
```

Validation des beans – JSR 303

- La JSR 303 est applicable aux beans quel que soit le contexte dans lequel ils sont utilisés. Nous la mettrons en œuvre ici dans un contexte Web mais ce n'est absolument pas une obligation.
- Spring MVC permet d'effectuer une validation des backing beans de manière automatisée. Pour ce faire il suffit de préfixer le bean à valider par l'annotation `@Valid` (Cette annotation s'applique à tous les @ModelAttribute qui suivent).
  - Spring MVC référencera automatiquement les erreurs de validation dans une instance de `BindingResult`. Cette instance peut être exploitée dans le contrôleur, il suffit d'ajouter un argument de ce type à la méthode après le dernier `@ModelAttribute`.

```
@RequestMapping("/ajoute")
    public String add(@Valid @ModelAttribute("acteur") Acteur acteur,
BindingResult result){

System.out.println(result.hasErrors());

        return "modif-form";
    }
```

Validation des beans – JSR 303

- L'instance de `BindingResult` va pouvoir également être exploitée dans les vues afin bien évidemment de présenter à l'utilisateur ses éventuelles erreurs de saisie.
- Dans l'exemple ci-dessous, nous avons choisi de reporter à côté de chaque champ de formulaire incriminé les éventuelles erreurs. La balise spécifique `<form:errors>` va permettre d'exploiter le contenu du `BindingResult` :

```
<form:form commandName="acteur" action="update.do">

    Nom <form:input path="nom" /><form:errors path="nom"/><br/>
    Prenom <form:input path="prenom" /><form:errors path="prenom"/><br/>
    <input type="submit" value="valider"/>


</form:form>
```

Validation des beans – JSR 303

- Il existe plusieurs façons de spécifier le validateur qui va servir à traiter les annotations. Spring propose un certains nombre de mécanismes qui lui sont spécifiques (Cf : `@InitBinder, WebdataBinder, LocalValidatorFactoryBean`)
- Cependant, plutôt que d'utiliser ces mécanismes spécifiques nous allons utiliser l'implémentation de Validator par défaut que Spring va trouver dans le classpath (ici le Validator Hibernate). Cette opération peut être réalisée simplement par l'ajout de la balise `<mvc:annotation-driven/>`, mvc: étant le préfixe d'un nouveau namespace.
  - Cette balise déclenche de nombreux comportements implicites (Support implicite des `@RequestMapping`, « content negociation »...).

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">

<mvc:annotation-driven/>
```

TP n°7

- Voir document dédié

@SessionAttributes

- Pour manipuler l'objet `HttpSession`, il suffit de le positionner en paramètre de la méthode cible du contrôleur.
- On peut également exploiter directement un objet issu de la session. L'annotation `@SessionAttributes` placée au niveau de la classe permet d'indiquer quels attributs seront stockés en session plutôt qu'en request.
  - Attention : Un attribut positionné en session doit alors impérativement exister lorsqu'il est transmis à la méthode

```
@Controller
@RequestMapping("/gestioncaddie")
@SessionAttributes("caddie")
public class HelloController {

    @RequestMapping("/create")
    private void createCaddie(Model model) {
        Caddie c=new Caddie();
        model.addAttribute("caddie",c);
     }
    @RequestMapping("/add")
    private void add(@ModelAttribute("caddie") Caddie c) {
        c.ajouteLivre(unLivre);
     }
```

Scopes pour le Web

- Les beans que nous venons de manipuler et qui étaient déposés en request ou en session sont en réalité pris en charge par le conteneur avec un scope request et un scope session, tout comme nous avions des beans en scope prototype ou singleton.
- Dans les applications web, les scopes supplémentaires dont on dispose sont :
  - request
  - session
  - globalSession (session globale au niveau des portlets)

Scopes pour le Web

- Nous aurions tout aussi bien pu déclarer dans la configuration XML d'autres beans dans ces scopes Web
- Avec Spring MVC, il suffit de donner à l'attribut scope des beans du fichier de configuration la valeur souhaitée : request, session etc...
- Pour récupérer un bean quelque part il suffit de l'injecter (`@Resource`)
  - On peut également manuellement interagir avec le `WebApplicationContext` en injectant dans le bean qui le nécessite la valeur d'une propriété de type `WebApplicationContext`

```
@Resource
private WebApplicationContext context;
```

Mixage des scopes

- Le problème de mixage de scopes rencontré avec singleton et prototype se pose également lorsqu'un bean en scope session fait référence à un Bean de scope request
- Le premier bean étant instancié en début de session, aucune chance de voir le bean de scope request se mettre à jour à chaque requête.
- Le bean de scope request aura « implicitement » la portée session !
- Attention : Le mixage des scopes traditionnels et des scopes Web est une toute autre affaire. Dans les exercices réalisés, le bean de type `Controller` en scope singleton n'avait pas de référence avec les beans en scope Web. Si l'on avait injecté une propriété en scope Web, nous aurions obtenu une erreur.
  - Résoudre le problème est possible mais passe par des considérations aop (voir `<aop:scoped-proxy>`)

```
<bean id="oss117" class="com.jnesis.Caddie" scope="session">
    <aop:scoped-proxy/>
</bean>
```

TP n°8 [Selon avancement]

- Voir document dédié

Controllers Restful

- Les controllers Spring MVC sont naturellement Restful. Cela signifie qu'ils fournissent sans modification des Web Services REST.
- Les méthodes de controllers peuvent bénéficier :
  - De l'annotation `@ResponseBody` sur les objets retournés
  - De l'annotation `@RequestBody` sur les objets fournis
- Ces annotations indiquent que des objets peuvent être sérialisés en chaîne de caractère vers le corps de la réponse et le corps de la requête peut être transformés en objets Java
  - Le format est alors la plupart du temps XML ou JSON

Controllers Restful

- Par défaut, le fait d'utiliser ces annotations n'a pas d'impact
- L'activation de cette fonction est lié à la mise en œuvre d'une configuration de ce que l'on appelle la « content negociation », sujet étudié plus loin dans le chapitre « Remoting »
- Néanmoins, on peut éviter cette fastidieuse configuration en ajoutant la balise `<mvc:annotation-driven>` celle déjà exploitée pour activée la validation des beans.
- Grâce à cette balise, Spring MVC va rechercher s'il existe dans le classpath des librairies susceptibles d'effectuer ces transformations
  - C'est le cas de Jackson, la librairie la plus largement utilisée

TP n°9 [Selon avancement]

- Voir document dédié