I - Technologies pour framework

1. Nouveau framework (généralités)

1.1. Intérêts récurrents liés à un nouveau framework

- Automatiser le répétitif (contrôles, liaisons , collaborations, ...)
- Gagner en productivité (si réutilisation fréquente du framework)
- Gagner en robustesse (si framework éprouvé bien au point)
- Structurant (beaucoup de composants partagent la même structure et la même logique) ---> maintenance à priori plus aisée.
- Intellectuellement enrichissant pour le concepteur qui met au point le framework.

1.2. Inconvénients fréquents d'un nouveau framework

- Yet Another Framework (risque de réinventer la roue si framework existant proche)
- Tout framework tend à imposer certaines structures (héritage ou ....) . Ceci peut éventuellement s'avérer bloquant (manque d'ouverture)
- Un framework mal documenté (qui n'explique pas exactement ce qu'il automatise) est un mauvais framework (incompréhensible , ...)
- Pour l'utilisateur d'un framework : soit intellectuellement enrichissant si l'on cherche à étudier le fonctionnement interne du framework , soit intellectuellement appauvrissant si on se contente de coder les sous composants et les paramétrages attendus par mimétisme (copier/coller d'autres exemples) ; manque de liberté ; initiative limitée .

2. Technologies pour Framework

2.1. Analyse d'un fichier de configuration xml

La plupart des frameworks puisent leurs configuations dans des fichiers xml (ex: springConf.xml , struts-config.xml, faces-config.xml , ...) .
Un nouveau framework peut procéder de la même façon et il aura donc besoin de lire et de convenablement interpréter l'arborescence des balises d'un fichier xml.

Technologies classiques pour le parsing XML (en java):

- SAX
- ***DOM*** (et variantes jdom , dom4j)
- ***JAXB2***
- ***stax*** (stream api for xml)

Configuration xml ===> configuration orienté objet en mémoire ===> application/exécution.

Alternatives à la configuration xml :

- **fichiers ".properties"** (plus simples à analyser mais pas arborescents)
- **annotations** (ex: *@Table(name="xxx")*) présentes dans certaines parties du code java des composants (à analyser par ***introspection*** lors de l'exécution ou bien via ***apt*** [annotation processing tool])

2.2. Introspection et méta classe java.lang.Class

De nombreux frameworks automatisent certains traitements en analysant la structure des classes java de l'application (mécanisme d'introspection).

Rappels sur l'introspection

Introspection

Le package **java.lang.reflect** permet d'effectuer une **introspection** des classes java. L'introspection consiste à ***demander aux classes de dresser la liste de leurs attributs et méthodes***.

Cette **faculté d'auto-analyse** qu'offre les mécanismes internes du langage Java est pour l'instant inexistante en C++ .
➔L'introspection est un **gros point fort de Java** .
➔Ceci ***permet d'automatiser entièrement des opérations de bas niveaux (sauvegarde / restauration de valeurs, ...).***

```
import java.lang.reflect.*;

Class c = Class.forName(nomClasse);
Field[] tabChamps = c.getDeclaredFields();
Method[] tabMethods = c.getDeclaredMethods();
 + boucle "for(i=0; i < tabXxxx.length; i++) ...
```

NB: depuis le jdk1.5 , il est également possible de récupérer dynamiquement à l'exécution du programme la liste de toutes les annotations (de rétention adéquate) (ex: @Id , ...).

Analyse d'une structure pour recopier des données

....

// La Classe **Field** (représentant un attribut d'une classe) comporte tous les éléments nécessaires
// pour récupérer la visibilité , le type , le nom et la valeur d'un attribut.

// On peut donc assez facilement balayer par boucle tous les attributs de façon à automatiser certains
// traitements (recopie de valeur , affectaction de valeur , ...)

....

Pour simplifier la manipulation des propriétés d'une instance java par introspection, on pourra éventuellement s'appuyer sur les classes du package "***org.apache.commons.beanutils***"(de ***commons-beanutils.jar***) .

Analyse et/ou sélection d'une opération pour l'invoquer

```
Class implClass = Class.forName(className);
  // ou bien implClass = objXxx.getClass();
String nomMethode="validate"; // méthode à invoquer
Method[] tabMeth = implClass.getMethods();
for(i=0;i<tabMeth.length;i++)
        if(tabMeth[i].getName().equals(nomMethode))
                { index=i; break; }
if(index>0)
  tabMeth[index].invoke(objXx, yyyValue);  //approfondir si besoin java.lang.Method et invoke
                                           // via javadoc du jdk
...
```

Prise en compte d'une annotation à l'exécution (runtime):

ex: code de la nouvelle annotation @A_valider

```
....
@Documented
@Retention(RUNTIME)
public @interface A_valider {
      /** Message si invalide */
      String value();
     ...
     /** type à valider (défaut : STRING). */
      TypeData  data_type() default  TypeData.STRING;
      /** Enumération des différents niveaux de criticités. */
      public static enum  TypeData { INT, DOUBLE, STRING };
}
```

(Remarque: l'information data_type est simplement içi montrée pour la syntaxe (avec enum) . Cette information n'est pas très utile dans un cas réel car l'introspection classique est déjà capable de récupérer l'information de type . Dans un cas de validation réel , on pourra plutôt placer des informations de type "min" , "max" , ...).

Utilisation (déclarative):

```
import xxx.A_valider;
class Xxx {
@A_valider(value="doit être un entier" , data_type=INT)
int age;

@A_valider(value="doit être une chaine")
String nom;
...
```

```
}
```

Prise en compte (test effectué par le framework):

```
// Instanciation de l'objet:
Xxx objet = new Xxx();

// On récupère la classe de l'objet :
Class<Xxx> classInstance = objet.getClass();

// On regarde si la classe possède une annotation :
A_valider annotation = classInstance.getAnnotation(A_valider.class);

if (annotation!=null) {
      System.out.println ("MonAnnotation : " + annotation.value() );
      }
```

Instanciation dynamique (sans new explicite):

```
String className = .....; // nom complet avec package (ex: "java.util.Date")
Class implClass = Class.forName(className);
Object objRes = implClass.newInstance();
....
```

2.3. Autres technologies diverses pour les frameworks

Décorateur / Proxy / Intercepteur

Beaucoup de frameworks automatisent certains éléments en:

- introduisant/interposant un nouvel objet intermédiaire entre le client appelant et le service appelé.
- cet intermédiaire (appelé "intercepteur" , "décorateur" , "enveloppe" ou "proxy") reprend la même interface que le service d'origine (mêmes méthodes publiques exposées) et ces nouvelles versions des méthodes (ainsi directement appelées par le client) vont en général :
  - introduire (avant et/ou après) de nouvelles fonctionnalités (ex: logs , tests , automatisme xy, ...)
  - rappeler en interne les méthodes de mêmes noms sur le service d'origine
- placer/relier le nouveau composant intermédiaire de façon idéalement transparente entre le client et le service d'origine (via une fabrique à Proxy ou AOP ou ....).

Programmation par aspects

Certains éléments de la programmation par aspects peuvent être utiles à la mise en oeuvre d'un framework (ex: ajout de fonctionnalités --> décorateur/proxy/intercepteur, ...).

Concrètement , un aspect supplémentaire nécessite une technologie adéquate (ex: Spring AOP , aspectJ, ...) . En d'autres termes le nouveau framework doit assez souvent être construit à partir d'un

premier framework (ex: Spring ou ...) pour bénéficier de AOP.

Initialisation coté java/web via Listener(s)

Dans beaucoup de frameworks "java/web" , on a assez souvent besoin de déclencher certaines initialisations dès le chargement de l'application dans le conteneur Web (Tomcat ou ....).

Un **Listener** (de l'api des ***Servlet***) est tout à fait approprié pour remplir cette tâche.

Rappels:

Certains ***événements liés à une application WEB*** (ou plus exactement à l'objet central "ServletContext") peuvent être gérés au sein d'objets appelés "**Listener**" .
Ceci permet essentiellement de ***déclencher automatiquement certains traitements*** aux moments des ***chargement/initialisation*** et ***arrêt/déchargement*** d'une ***application WEB***.

Interfaces événementielles (javax.servlet)	***Descriptions***
ServletContextListener	objet "application" (ServletContext) tout juste créé ou bien sur le point d'être supprimé.
ServletContextAttributeListener	Attribut ajouté,supprimé ou modifié sur l'objet application (ServletContext)

NB:

Une classe d'objet "***Listener***" doit ***implémenter l'interface événementielle adéquate*** et son code compilé doit être placé dans **WEB-INF/classes** ou bien dans un des **"...jar"** de **WEB_INF/lib**.

D'autre part, un "***Listener***" (gestionnaire d'événements) doit être déclaré au sein du fichier ***WEB-INF/web.xml*** pour qu'il soit pris en compte:

```
<web-app ...>
<display-name>MyListeningApplication</display-name>
<listener>
    <listener-class>mypackage.MyListenerClass</listenerclass>
</listener>
<listener>
      <listener-class>mypackage.MyOtherListenerClass</listener-class>
</listener>
<servlet>...</servlet>
...
</web-app>
```

Exemple:

```
package mypackage;

import javax.servlet.ServletContext;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
```

```
public class MyListenerClass implements ServletContextListener {

        public void contextInitialized(ServletContextEvent e) {

      // initialisation au chargement/démarrage de l'application WEB
                ServletContext application = e.getServletContext();
                Integer objCompteur = new Integer(1);
                application.setAttribute("compteur",objCompteur);
        }

        public void contextDestroyed(ServletContextEvent e) {

      //  terminaison lors de l'arrêt de l'application WEB
                ServletContext application = e.getServletContext();
                Integer objCompteur = (Integer) application.getAttribute("compteur");
                System.out.println("compteur:" + objCompteur.intValue());
        }

}
```

Enrichissement de la partie html/javascript via des Filtres

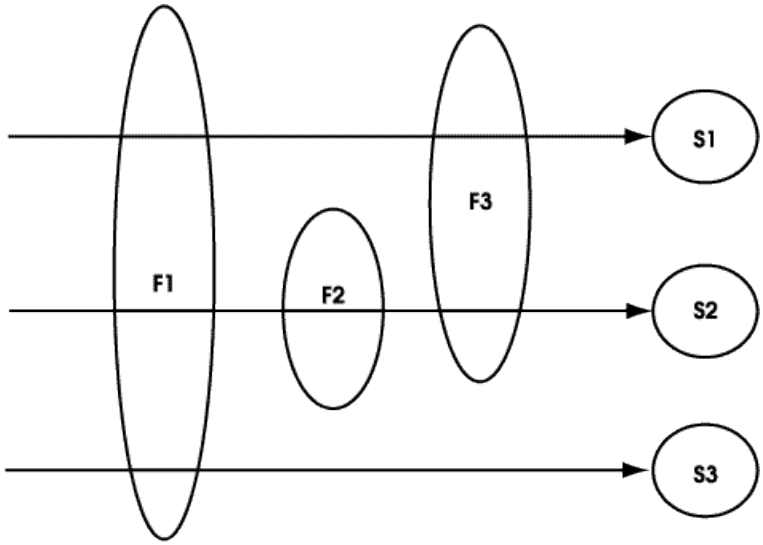
Notion de filtre:
Un filtre est un élément supplémentaire qui s'insère en tant qu'enveloppe au niveau de la chaîne d'exécution des servlets et qui peut modifier la requête et/ou la réponse Http.

Ceci peut être utile pour effectuer l'une des tâches suivantes:

- Effectuer des conversions (images, données, ...)
- Gérer le cryptage des données
- Effectuer des transformations XSLT.
- Effectuer des compression & décompression (gzip)
- Rajouter automatiquement des entêtes , des compteurs , des stats, ...
- Ajouter des éléments javascripts (menus , validations, ...)

Insertion d'un filtre (Filter Mapping de web.xml):
Un filtre n'est jamais directement mentionné dans une url , il est simplement associé à certaines url en tant qu'élément supplémentaire (s'activant avant en tant qu'enveloppe) :

Nb: l'ordre des filtres correpond à l'ordre des balises <filter-mapping> de web.xml.




```
<web-app>
<filter>
   <filter-name>XSLTFilter</filter-name>
   <filter-class>XSLTFilter</filter-class>
 </filter>
<filter>...</filter>
  <filter-mapping>
    <filter-name>XSLTFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
  <filter-mapping> ... </filter-mapping>
 <servlet> ... </servlet>  <servlet-mapping> .... </servlet-mapping>...
<web-app>
```

Programmation d'une classe de Filtre:

```
public final class XXXFilter implements Filter {
   private FilterConfig filterConfig = null;

   public void init(FilterConfig filterConfig)
      throws ServletException { this.filterConfig = filterConfig; }

   public void destroy() { this.filterConfig = null; }

   public void doFilter(ServletRequest request,
      ServletResponse response, FilterChain chain)
      throws IOException, ServletException {
           //... code du filtre ....
        chain.doFilter(request,response); // déclenche l'élément
                          // suivant (autre filtre ou servlet , ...)
        }
}
```

De façon à pouvoir modifier la réponse générée par le code du servlet (élément suivant de la chaine) , il faut invoquer **chain.doFilter(request,wrapper);** où **wrapper** est une enveloppe autour de response dont la méthode **getWriter**() retourne un flux sur un paquet d'octets qui ne sera pas directement renvoyé mais qui sera retraité par la fin du filtre actuel:

```
public class CharRespWrapper extends   HttpServletResponseWrapper {
```

```
  private CharArrayWriter output;
   public String toString() {  return output.toString(); }
   public CharResponseWrapper(HttpServletResponse response)
                { super(response); output = new CharArrayWriter(); }
  public PrintWriter getWriter() {return new PrintWriter(output);}
}
```

```
...
PrintWriter out = response.getWriter();
CharRespWrapper wrapper = new
CharRespWrapper( (HttpServletResponse)response);
chain.doFilter(request, wrapper);
out.write("...." + wrapper.toString() + "...");
```

2.4. JSF en mode dynamique (avec introspection)

<h:dataTable **binding**="#{xxxxCRUD.***dynamicDataTable***}" /> coté (jsp / xhtml) et

```
package generic.web.fwk.dynamic_jsf;

import java.lang.reflect.Field;
import java.util.List;

import javax.el.MethodExpression;
import javax.el.ValueExpression;
import javax.faces.component.UIComponent;
import javax.faces.component.UIParameter;
import javax.faces.component.html.HtmlColumn;
import javax.faces.component.html.HtmlCommandLink;
import javax.faces.component.html.HtmlDataTable;
import javax.faces.component.html.HtmlInputText;
import javax.faces.component.html.HtmlOutputText;
import javax.faces.component.html.HtmlPanelGroup;
import javax.faces.context.FacesContext;

public class MyDynamicJsfGenerator {

        public HtmlOutputText genHtmlOutputText(String texte){
                HtmlOutputText htmlOutputText= new HtmlOutputText();
                htmlOutputText.setValue(texte);
```

```
		return htmlOutputText;
	}

	public HtmlOutputText genHtmlOutputText(String exprValue,
					Class exprType/* Sting.class if null*/,
					String id /* may be null*/){
		HtmlOutputText htmlOutputText= new HtmlOutputText();

htmlOutputText.setValueExpression("value",createValueExpression(exprValue,exprType));
		if(id!=null)
			htmlOutputText.setId(id);
		return htmlOutputText;
	}

	public HtmlDataTable populateDataTableFromBeanIntrospection(Class
metaClass,String mbeanName){
		HtmlDataTable dynamicDataTable=new HtmlDataTable();
		dynamicDataTable.setBorder(1);
		dynamicDataTable.setVar("entityDto");
		dynamicDataTable.setValueExpression("value",
			createValueExpression("#{"+mbeanName+ ".resultList}",
				List.class));

		for(Field f : metaClass.getDeclaredFields()){
			if( ! java.lang.reflect.Modifier.isStatic(f.getModifiers()) ){
				String fName=f.getName();

				HtmlColumn column = new HtmlColumn();
				column.setId("col"+fName);// ID need for dynamic Column
				column.setHeader(genHtmlOutputText(fName));


column.getChildren().add(genHtmlOutputText("#{entityDto."+fName+"}",
							String.class, /*id*/
"text_"+fName));
				dynamicDataTable.getChildren().add(column);
			  }
			}
		// colonne supplémentaire "link vers détails":
		HtmlColumn columnDetails = new HtmlColumn();
		columnDetails.setId("colDetails");// ID need for dynamic Column
		columnDetails.setHeader(genHtmlOutputText("selection"));

		HtmlCommandLink link = new HtmlCommandLink();
```

```
                link.setId("link_details");
                link.setValue("details");
                link.setActionExpression(
                                createMethodExpression("#{"+mbeanName+ ".doSelect}",
                                                String.class,new Class[0]));
                UIParameter param = new UIParameter();
                param.setName("paramId");
                param.setValueExpression("value",
                                createValueExpression("#{entityDto.id}",
                                                Object.class));
                param.setValueExpression("binding",createValueExpression("#{"+mbeanName+
".paramId}",
                                UIParameter.class));
                link.getChildren().add(param);
                columnDetails.getChildren().add(link);
                dynamicDataTable.getChildren().add(columnDetails);
                return dynamicDataTable;
        }

        public HtmlPanelGroup populateFromBeanIntrospection(Class metaClass,String
mbeanName,String entityDtoName){

                HtmlPanelGroup dynamicFormGroup = new HtmlPanelGroup();
                UIComponent uIComponent;
                for(Field f : metaClass.getDeclaredFields()){
                  if( ! java.lang.reflect.Modifier.isStatic(f.getModifiers()) ){
                        String fName=f.getName();

dynamicFormGroup.getChildren().add(/*label*/genHtmlOutputText(fName));

                        uIComponent=new HtmlInputText();
                        uIComponent.setValueExpression("value",
                                        createValueExpression("#{" + mbeanName +"."
+entityDtoName +"."+fName+"}",
                                                                        String.class));
                        dynamicFormGroup.getChildren().add(uIComponent);

                        // linebreak construction
                        HtmlOutputText  linebreak = new HtmlOutputText();
                        linebreak.setValue("<br/>");   linebreak.setEscape(false);
                        dynamicFormGroup.getChildren().add(linebreak);
                        }
                }
                return dynamicFormGroup;
```

```
        }


        // Helper ----"createValueExpression"----------------------------------------------
         public ValueExpression createValueExpression(String valueExpression, Class<?>
valueType) {
        FacesContext facesContext = FacesContext.getCurrentInstance();
        return facesContext.getApplication().getExpressionFactory().createValueExpression(
                facesContext.getELContext(), valueExpression, valueType);
        }
        // Helper ----"createMethodExpression"----------------------------------------------
         public  MethodExpression createMethodExpression(String valueExpression, Class<?>
valueType,Class<?>[] argTypes) {
                        FacesContext facesContext = FacesContext.getCurrentInstance();
                        return
facesContext.getApplication().getExpressionFactory().createMethodExpression(
                                facesContext.getELContext(), valueExpression,
valueType,argTypes);
                        }

}
```