L'arboretum de St André

Description du thème

Propriétés	Description
Intitulé long	Exploitation du framework JSF et de l'API JPA au travers d'une série d'applications web basées sur MVC2
Formation concernée	BTS SIO 2ème année, option SLAM
Matière	SLAM 4
Présentation	Découvrir en quelques étapes les principes et les avantages de ces outils au travers d'exemples fournis à interpréter et exécuter, voire à compléter.
Notions	A4.1.4 Définition des caractéristiques d'une solution applicative A4.1.6 Gestion d'environnements de développement et de test A4.1.7 Développement, utilisation ou adaptation de composants logiciels Savoirs associés • Caractéristiques d'un framework • Persistance et couche d'accès aux données, technologies et techniques associées
Transversalité	SLAM 3
Pré-requis	Langage objet, IDE Eclipse, bases de données
Outils	Eclipse Juno JavaEE, Glassfish Open Source, MySql, PhpMyAdmin
Mots-clés	MVC2, Glassfish, Eclipse, JSF, JPA, JEE6
Durée	Quatre fois deux heures.
Auteur(es)	Jean-Philippe Pujol avec la relecture attentive de Gaëlle Castel
Version	v 1.0
Date de publication	Mars 2013

Contexte

Disposant d'une surface de plus de deux hectares l'arboretum de Saint André est situé dans le sud-ouest de la France. Il propose l'observation d'arbres de toutes natures et de toutes provenances à ses visiteurs.

Géré par une association à visées écologiques, l'arboretum montre le cycle de vie de nombreuses essences en apportant de nombreux renseignements sur des détails botaniques souvent mal perçus par les profanes.

La présentation de ces arbres en quantité de plus en plus importante était jusque là dévolue à des jardiniers dont la compétence, certes étendue, ne permet plus aujourd'hui de répondre avec suffisamment d'efficacité aux questions posées par les visiteurs. Le recours à une solution informatique mémorisant les caractéristiques propres à chaque type d'arbre (caractéristiques, origine, dénomination, etc.), mais aussi propre à chaque arbre (âge, emplacement, état général, historique, etc.) s'impose aux responsables de l'arboretum. À terme, ils envisagent même de recourir à une solution de réalité augmentée pour assister chaque visiteur tout en lui laissant une autonomie complète dans sa visite.

La société informatique dans laquelle vous êtes technicien est chargée de développer les briques de cette application. Dans un premier temps, nous envisageons la conception d'une application web dont les contraintes ont été énoncées :

- environnement JEE6 au travers du serveur d'application Glassfish ;
- mise en œuvre du framework JSF pour les pages web et de l'API JPA pour la couche de persistance des données ;
- utilisation de l'outil de développement Eclipse Juno Java EE ;
- persistance dans une base de données MySql.

Remarque : les commandes sont proposées dans un environnement Windows mais peuvent facilement être adaptées sous Linux.

Simulation

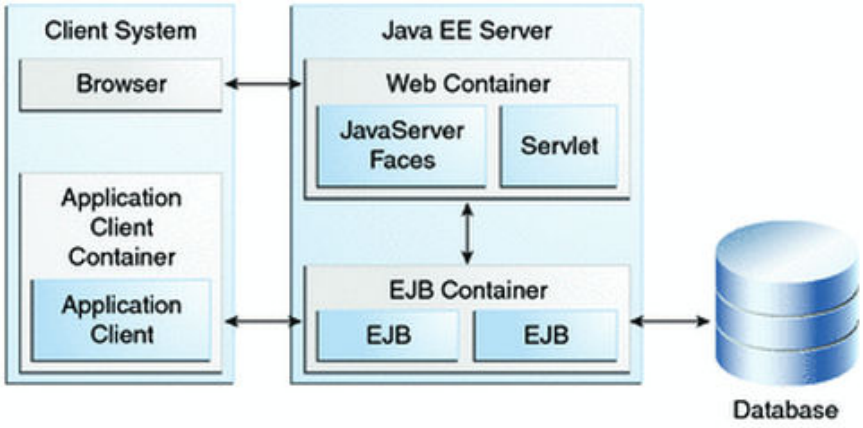
Dans le cadre de ce TP vous allez simuler le rôle du technicien qui découvre les caractéristiques des frameworks en mettant en œuvre des exemples fournis. Vous interpréterez le sens et le rôle de chaque composant en observant son fonctionnement et son rendu, voire en cherchant de la documentation détaillée.
Nous vous proposons diverses ressources utiles :
The Java EE 6 Tutorial [1]:
http://docs.oracle.com/javaee/6/tutorial/doc/docinfo.html
Langage JPQL [2] :
http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Querying/JPQL
JPA, Java File Libraries [3] :
http://www.java2s.com/Tutorial/Java/0355__JPA/Catalog0355__JPA.htm
JSF Tag reference, composants xhtml [4] :
http://www.jsftoolbox.com/documentation/help/12-TagReference/index.jsf

Principe

Le serveur d'application GlassFish est une implémentation de la plate-forme Java EE 6. Glassfish Open Source, utilisé ici, contient entre autres :

- EclipseLink, framework open source de mapping objet-relationnel supportant l'API de persistence JPA ;
- Derby, serveur de bases de données relationnelles qui n'est pas utilisé ici.

Le schéma ci-dessous (document Oracle [1] Part I Introduction, 1 Overview, Java EE 6 APIs) montre les relations entre les composants d'un développement web :



Le serveur d'application apparaît ainsi comme le maitre d'œuvre de l'architecture déployée.

Le framework JSF (Java Server Faces) qui va être mis en place est basé sur les composants : le développeur conçoit des classes dont le cycle de vie et l'état sont gérés par le serveur d'application, sans avoir à se soucier des requêtes transitant entre les objets.
JSF implémente le modèle MVC2, à savoir :

- le modèle : classes Java dont la persistance pourra être assurée par l'API JPA ;
- les vues qui sont des pages web XML contenant différents composants. Elles sont appelées « *Facelets* » ;
- un unique contrôleur, c'est la classe *FacesServlet* dont le développeur n'a pas à se préoccuper. Elle reçoit les requêtes des clients, contrôle, traite et détermine la vue à invoquer. Bien que totalement transparente dans le code, sa présence et son rôle doivent être bien perçus afin de comprendre les mécanismes sous-jacents.

L'appel aux services du serveur d'application se réalise au travers d'annotations mentionnées dans le code Java.

Installation d'un environnement de développement

Sur votre poste de travail, vous disposerez *in fine* de :

- Eclipse
- MySql
- GlassFish.

Eclipse

L' IDE (Integrated Development Environment) Eclipse est à télécharger à partir de :
http://www.eclipse.org/downloads/

Vous prenez *Eclipse IDE for Java EE Developers* (version Juno à ce jour, 228 MB). Une décompression du fichier téléchargé, de préférence à la racine du disque dur, suffit : le lancement se réalise à partir de l'exécutable *eclipse.exe*

MySql

Un serveur de base de données utilisant le port 3306 suffit ; il est possible de recourir à Xamp, WampServer ou tout autre package. L'utilisation du serveur web de ce dernier sera utile pour exécuter le client d'administration PhpMyAdmin, dans la mesure où le port utilisé est fréquemment le 80 (le 8080 étant réservé, entre autres, par GlassFish).

GlassFish

Le serveur d'application d'Oracle est disponible ici :
http://glassfish.java.net/fr/

Nous prenons la version zip Open Source :

Comment obtenir GlassFish 3.0.1 ?

Serveur GlassFish édition Open Source 3.0.1

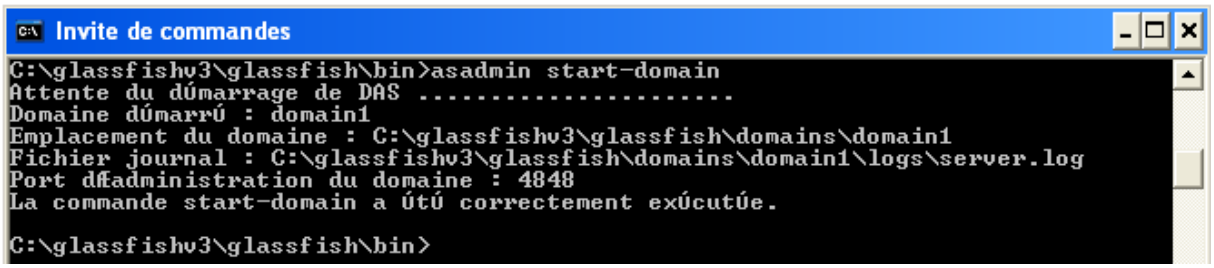
Distribution	Windows [1]	Taille (Mo)	Linux / Unix / Mac [2]	Taille (Mo)	Archive Zip [3]	Taille (Mo)
GlassFish 3.0.1 Édition Open Source **Profil complet**	glassfish-3.0.1-windows.exe (EN)	34	glassfish-3.0.1-unix.sh (EN)	54	glassfish-3.0.1.zip (EN)	77
	glassfish-3.0.1-windows-ml.exe (multilingue)	37	glassfish-3.0.1-unix-ml.sh (multilingue)	58	glassfish-3.0.1-ml.zip (multilingue)	85
GlassFish 3.0.1 Édition Open Source **Profil Web**	glassfish-3.0.1-web-windows.exe (EN)	34	glassfish-3.0.1-web-unix.sh (EN)	33	glassfish-3.0.1-web.zip (EN)	47
	glassfish-3.0.1-web-windows-ml.exe (multilingue)	37	glassfish-3.0.1-web-unix-ml.sh (multilingue)	35	glassfish-3.0.1-web-ml.zip (multilingue)	52

[1] : programme d'installation utilisant une interface graphique pour Windows. Peut être utilisé en mode silencieux.
[2] : programme d'installation utilisant une interface graphique pour Solaris, Linux et MacOS X. Peut être utilisé en mode silencieux.
[3] : fichier à télécharger indépendant de la plate-forme. Décompressez et lancez simplement domain1 par défaut.

Comme pour Eclipse, l'installation se limite à la décompression du fichier téléchargé, par exemple ici à la racine du disque dur.

Nous pouvons faire démarrer le serveur à partir d'une console après s'être positionné dans le bon répertoire, avec la commande suivante :

```
asadmin start-domain
```



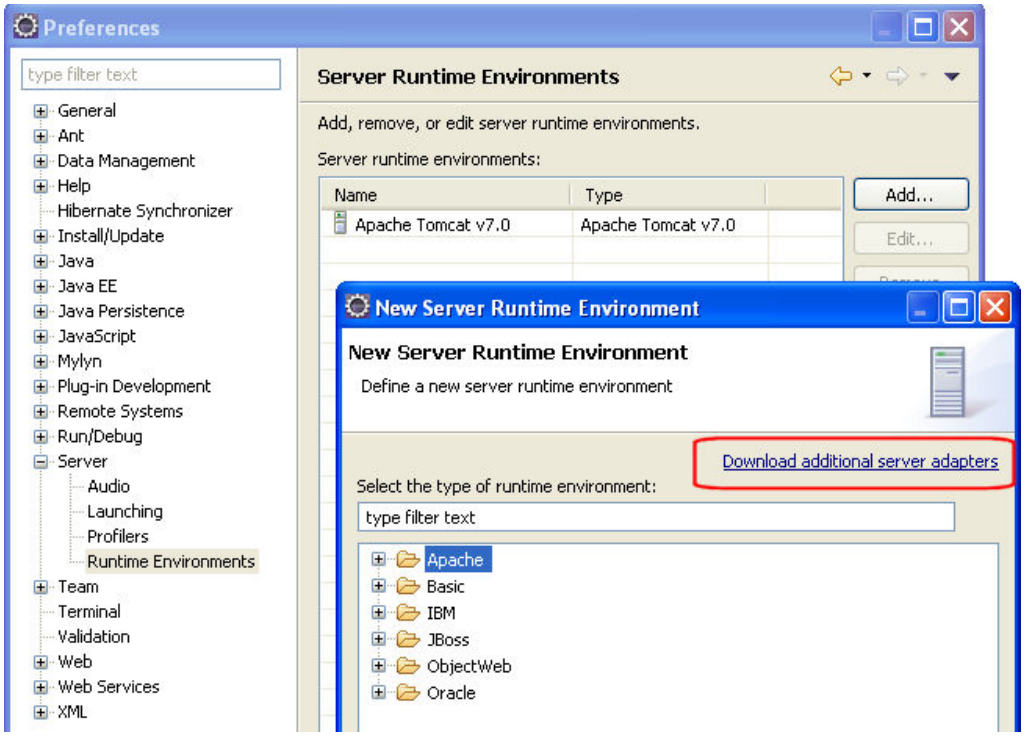
Nous accédons alors aux ressources publiées sur le port 8080 :

L'administration de Glassfish se fait quant à elle sur le port 4848.

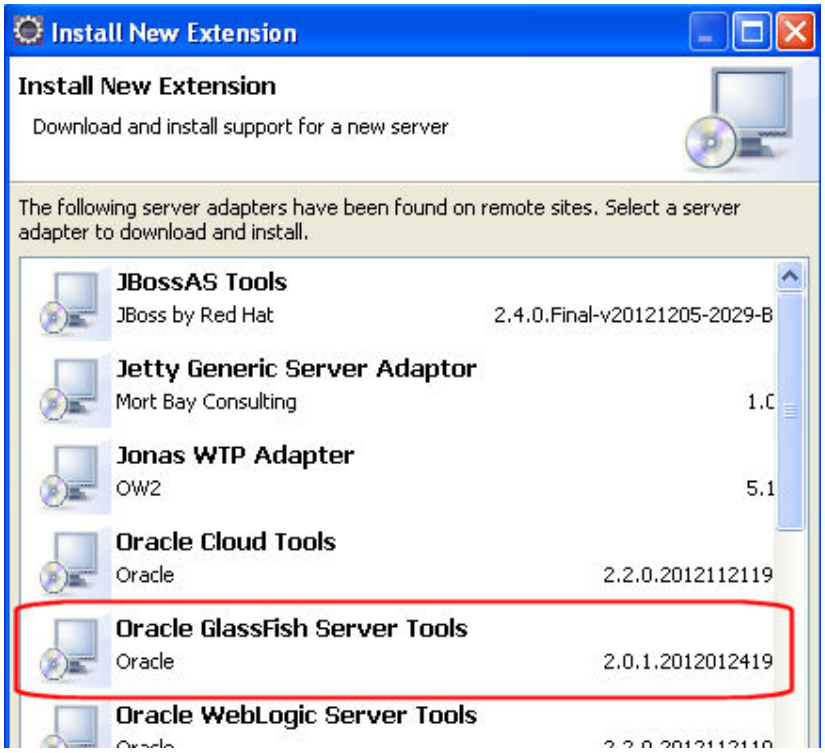
Intégration de GlassFish dans Eclipse Juno JavaEE

L'IDE Eclipse va nous permettre de développer des applications web en utilisant des assistants, mais aussi de déployer automatiquement celles-ci sur un serveur.
Pour cela ce serveur doit être référencé par Eclipse, ce qui n'est pas le cas par défaut.

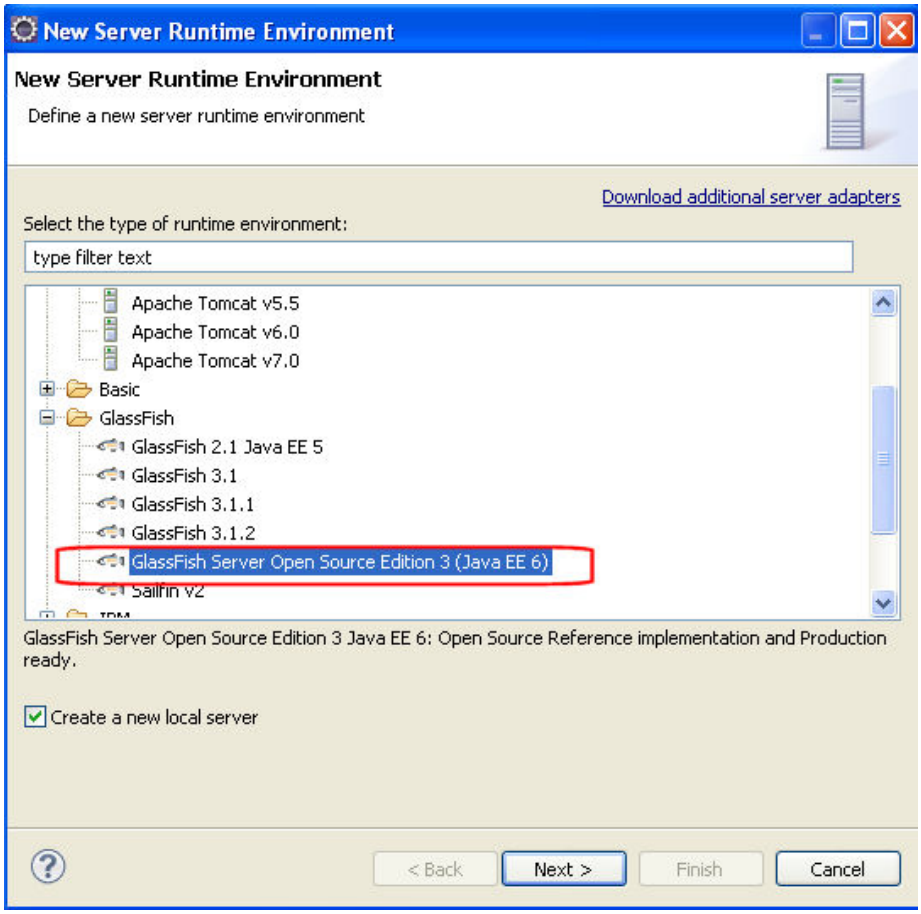
Si les serveurs GlassFish n'apparaissent pas dans la liste *Server Runtime Environment* accessible par *Window / Preferences*, cliquer sur *Download additional server adapters* :



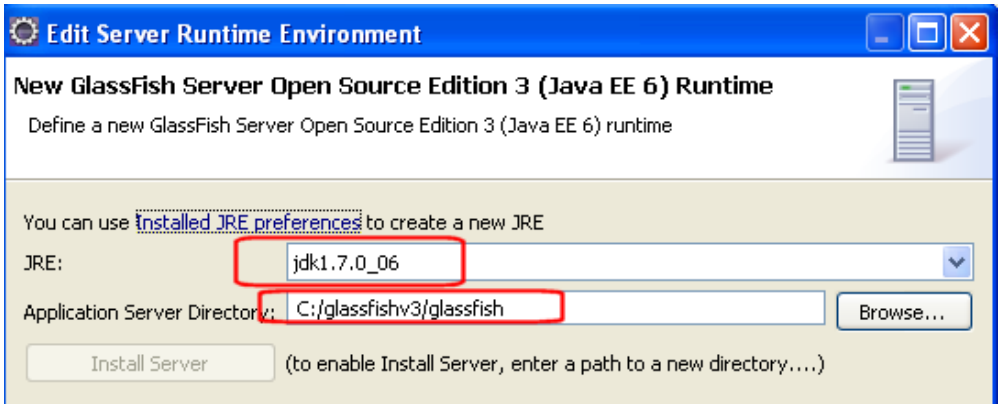
Choisir *Oracle GlassFish Server Tools* :



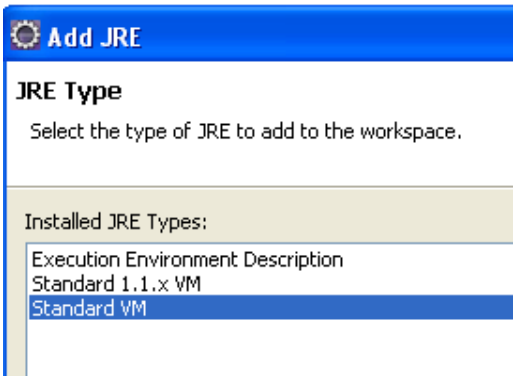
La liste précédente est maintenant enrichie des serveurs GlassFish et nous sélectionnons le type de celui qui vient d'être installé :



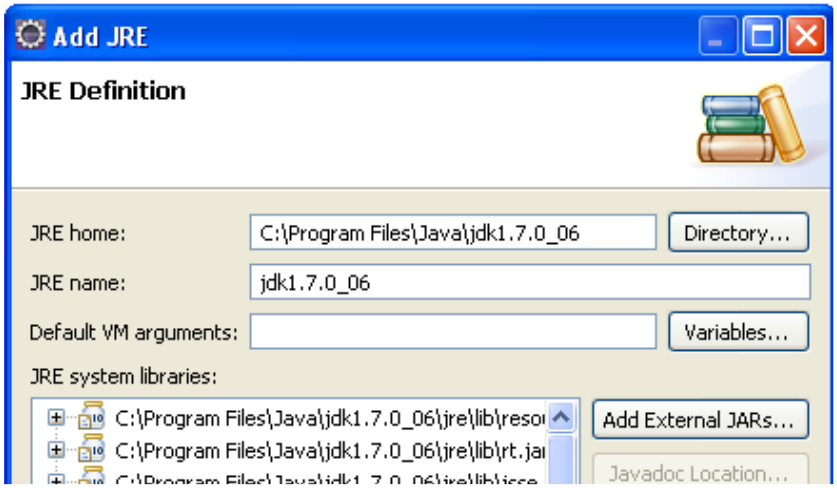
Nous cherchons son chemin d'accès choisi lors du décompactage :



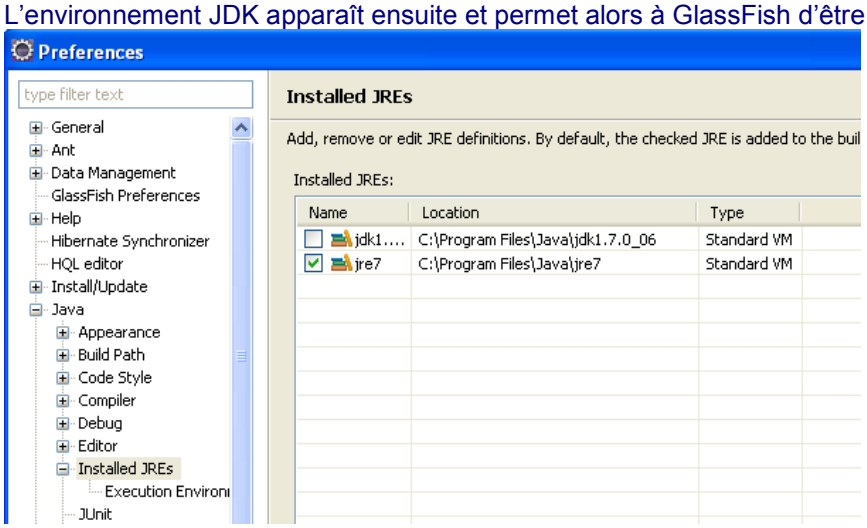
Remarque : GlassFish nécessite un environnement JDK pour la compilation des servlets. Si le JDK visible ci-dessus n'est pas sélectionnable dans la liste déroulante ci-dessus « JRE », il faut en faire la configuration. Dans *Window* / *Preference* / *Installed JRE*, nous cliquons sur *Add* puis nous sélectionnons « *Standard VM* » :

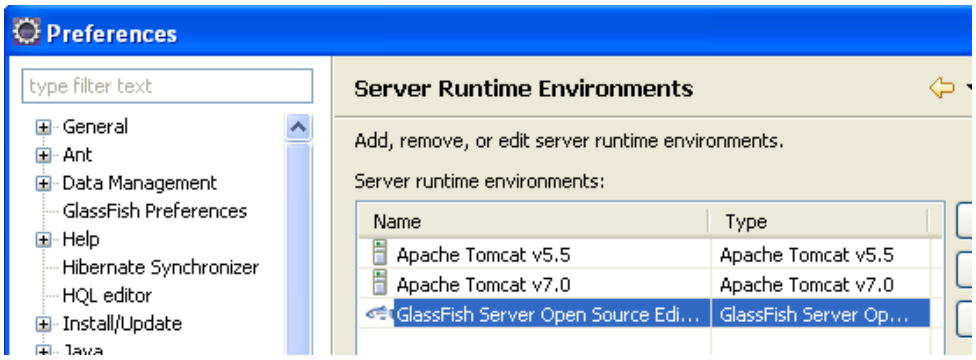


Le bouton *Directory* permet d'aller chercher un JDK, de préférence le plus récent :

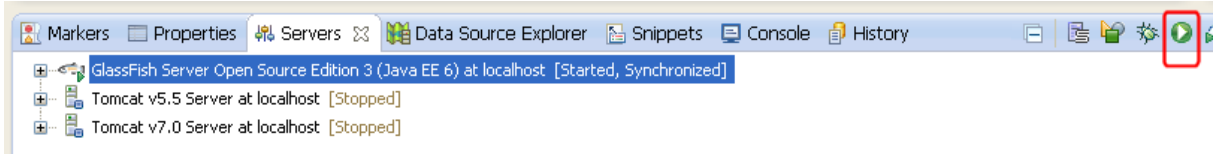


L'environnement JDK apparaît ensuite et permet alors à GlassFish d'être correctement enregistré :

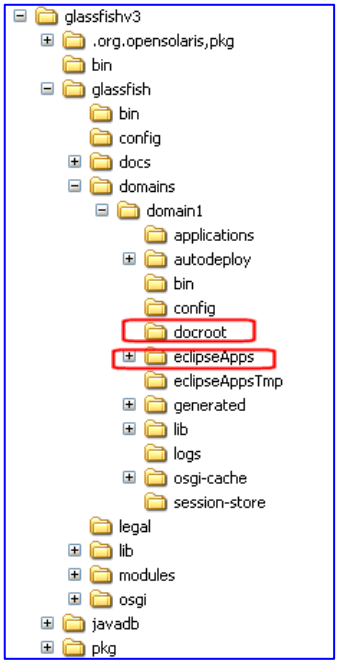




Après redémarrage d'Eclipse, nous trouvons dans la partie basse de l'interface d'Eclipse le serveur qui peut être démarré ou arrêté (boutons vert et rouge) :



Ici, il est montré démarré et nous pouvons tester une application web à l'aide de notre navigateur préféré (et non pas avec le navigateur intégré d'Eclipse).



Le répertoire dans lequel Glassfish a été décompressé montre le domaine par défaut (domain1) qu'il gère et dans celui-ci les répertoires :

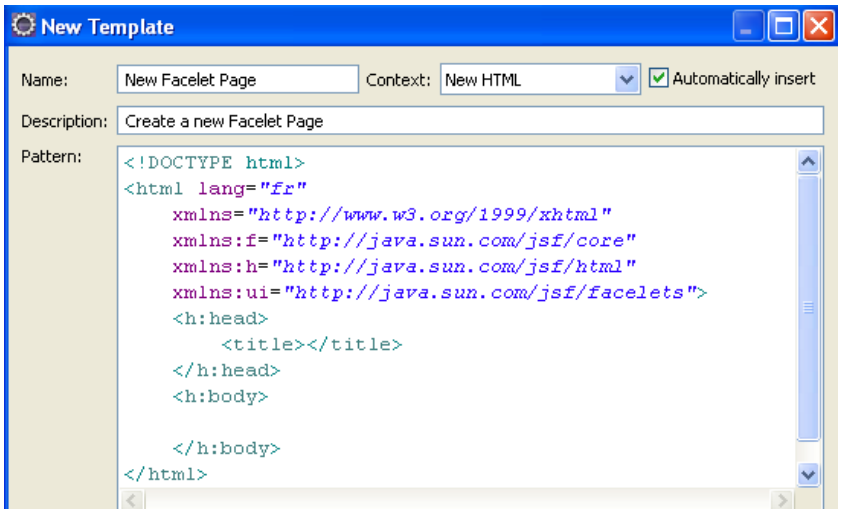
- **docroot** : répertoire de publication dans lequel il est possible de placer des pages HTML;
- **eclipseApps** dans lequel Eclipse installera les applications à exécuter ;
- **logs** : le répertoire des logs visible aussi depuis la console d'Eclipse.

Paramétrage initial d'Eclipse pour JSF

En vue de réaliser rapidement les pages web montrées dans les exemples qui suivent, il est pertinent de créer un modèle pour ces pages car celui-ci n'existe pas encore dans la version Juno.

Dans *Window* / *Preferences* nous sélectionnons *Web* / *HTML Files* / *Editor* / *Templates* et nous cliquons sur *New*. Pour rester dans l'esprit Eclipse, nous remplissons les champs comme dans la copie d'écran ci-dessous en y copiant le code standard des futures pages :

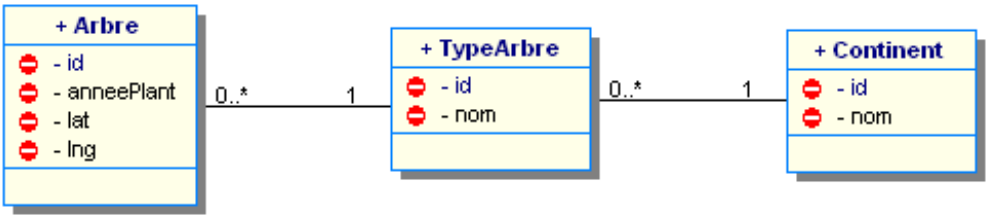
```
<!DOCTYPE html>
<html lang="fr"
   xmlns="http://www.w3.org/1999/xhtml"
   xmlns:f="http://java.sun.com/jsf/core"
   xmlns:h="http://java.sun.com/jsf/html"
   xmlns:ui="http://java.sun.com/jsf/facelets">
   <h:head>
      <title> </title>
   </h:head>
   <h:body>

   </h:body>
</html>
```



La conception des pages se limitera alors à l'écriture du code entre les balises <body> et </body>

Diagramme de classes

Dans le cadre de la simulation à réaliser, les traitements proposés dans les cas d'utilisation suivants exploiteront en partie ou en totalité les classes ci-dessous :



Les méthodes implémentées sont celles des *beans*, à savoir un constructeur vide ainsi que des *getters* et *setters* respectant la règle de dénomination usuelle.
Un type d'arbre est par exemple un Hévéa provenant du continent Amérique. Un arbre est une instance de type d'arbre, par exemple, un hévéa planté dans l'arboretum en 1987 et situé à un emplacement déterminé par sa latitude et sa longitude.
La gestion des arbres n'est pas abordée ici : elle pourra faire l'objet d'un développement spécifique ultérieur.

Cas d'utilisation

Premier cas : observation d'un *Facelet* et prise en main des outils

Ce premier cas consiste à saisir l'*id* et le *nom* d'un type d'arbre instanciant un objet de la classe TypeArbre, puis à en afficher le contenu.

Dans cette première partie, nous allons apprendre à créer un projet, des classes, démarrer et arrêter le serveur Glassfish, observer les logs, etc. : toutes ces manipulations devront être comprises car elles seront réalisées à l'identique dans les cas suivants.

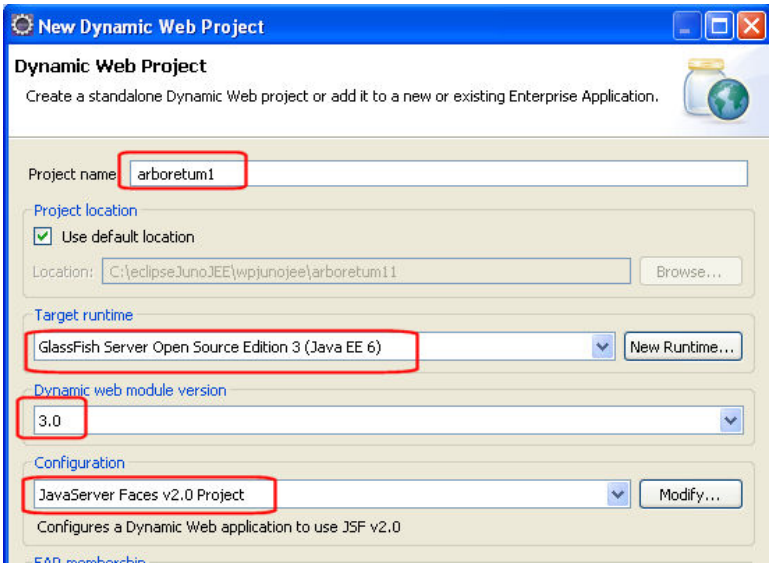
Résultat attendu

Dans ces copies d'écran le bouton *Enregistrer* permet d'enregistrer la saisie ; le lien *Autre saisie* permet de revenir au formulaire.

Nous nous limiterons ici à des saisies correctes : un entier pour l'*id* et une chaine pour le *nom*.

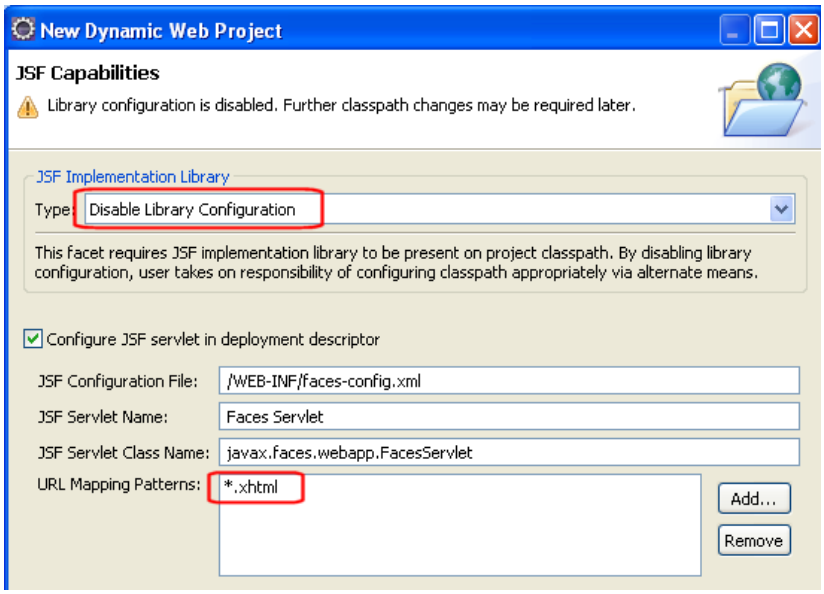
Création du projet

Vous créez un nouveau projet *Dynamic Web Project* selon les spécifications montrées dans cette copie d'écran :



Après avoir cliqué sur *Next*, le troisième écran suivant doit être modifié :

La librairie JSF est incluse dans le serveur Glassfish : nous invalidons donc le recours à une librairie externe (Disable Library Configuration).
Le choix de l'extension *.xhtml correspond à un usage. Il indique ici que toutes les requêtes clientes invoquant les fichiers portant cette extension seront obligatoirement traitées par l'unique contrôleur *FacesServlet*.



Création des classes et des pages web

Après création du projet, nous allons créer des packages dans le *src* de *Java Resources* afin d'organiser les classes selon leur responsabilité.

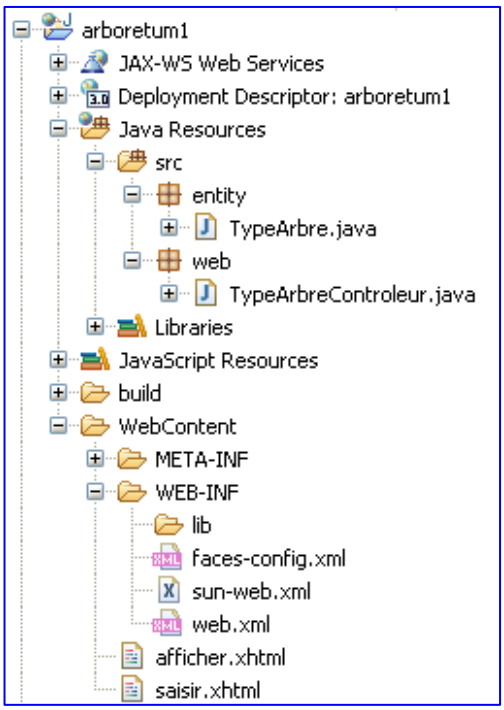
Ici nous adoptons la dénomination Oracle avec les packages *entity* et *web*.

Dans le package *entity*, nous plaçons les *java beans* correspondant aux classes métier (classes entités).

Dans le package *web*, nous plaçons un *java bean* spécifique chargé d'exposer les données des classes métiers aux pages **.xhtml**.

Dans le répertoire *WebContent*, nous plaçons les fichiers **.xhtml** invoqués depuis le navigateur du client.

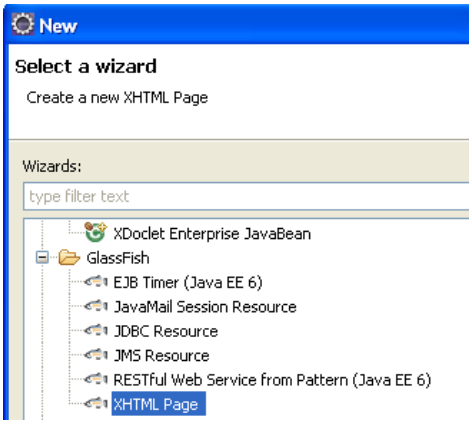
Le fichier **web.xml** est créé automatiquement ; nous pourrons le compléter pour faciliter le développement comme indiqué plus loin.



Travail à faire	
1.1	Créez la classe java TypeArbre comportant uniquement les propriétés *id* et *nom*. Générez un constructeur vide ainsi que tous les accesseurs.
1.2	Créez la classe java TypeArbreControleur à partir des données fournies ci-dessous, en particulier les annotations précédant la déclaration de la classe (importez les suggestions proposées par Eclipse en vérifiant à chaque fois le package d'origine en cas d'homonymie).
1.3	Créez les pages **saisir.xhtml** et **afficher.xhtml** (confer ci-dessous) en utilisant le modèle *New Facelet Page* créé précédemment. Complétez les balises <h:body> avec les copies d'écran ci-dessous.

```
13  @ManagedBean
14  @RequestScoped
15  public class TypeArbreControleur implements Serializable{
16      private static final long serialVersionUID = 1L;
17      private TypeArbre typeArbre ;
18
19      public TypeArbreControleur() {
20          this.typeArbre = new TypeArbre();
21      }
22      public TypeArbre getTypeArbre() {
23          return typeArbre;
24      }
25  }
```

Création d'une page **.xhtml** (*New* / *Other*) :



La page **saisir.xhtml** :

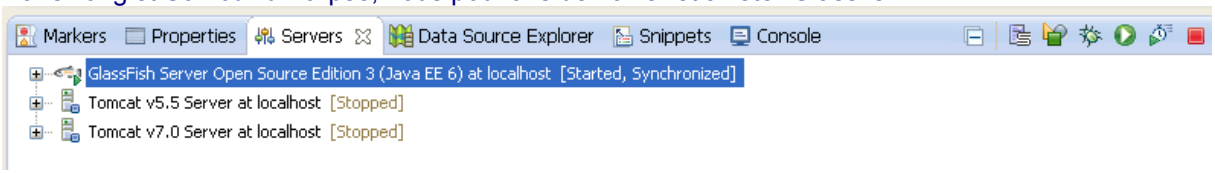
```
<!DOCTYPE html>
<html lang="en" xmlns="http://www.w3.org/1999/xhtml"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:ui="http://java.sun.com/jsf/facelets">
<h:head>
    <title>Saisie type arbre</title>
</h:head>
<h:body>
    <h:form>
        <h:panelGrid columns="2">
            <h:outputText value="Saisir code type" />
            <h:inputText value="#{typeArbreControleur.typeArbre.id}" />
            <h:outputText value="Saisir le nom du type" />
            <h:inputText value='#{typeArbreControleur.typeArbre.nom}' />
            <h:commandButton action="afficher" value="Enregistrer" />
        </h:panelGrid>
    </h:form>
</h:body>
</html>
```

Le contenu de la balise <h:body> (l'en-tête est toujours le même) de la page **afficher.xhtml** :

```
<h:body>
    <h:form>
        <h:outputText
            value="Le type dont l'id est #{typeArbreControleur.typeArbre.id}
                   est un #{typeArbreControleur.typeArbre.nom}" />
        <br />
        <h:commandLink action="saisir">
            <h:outputText value="Autre saisie" />
        </h:commandLink>
    </h:form>
</h:body>
```

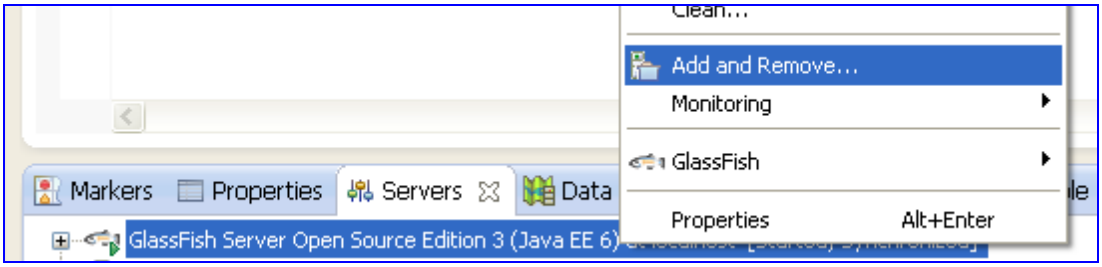
Démarrage du serveur Glassfish et déploiement de l'application

Dans l'onglet *Serveur* d'Eclipse, nous pouvons démarrer et arrêter Glassfish :

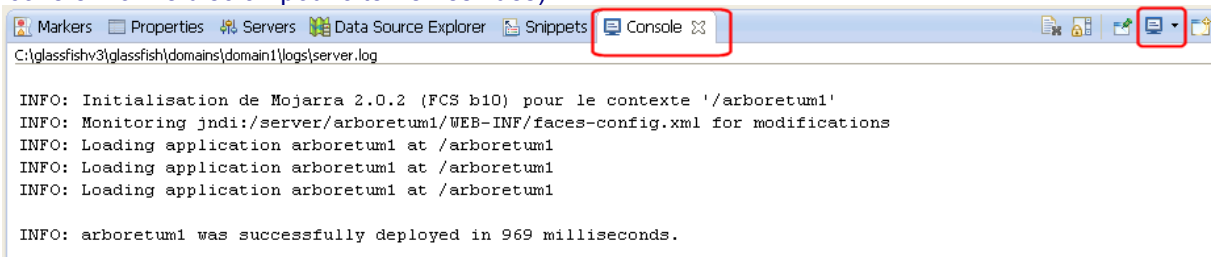


A priori, il n'y a pas d'intérêt particulier à l'arrêter pendant le développement car le déploiement des applications modifiées se fait automatiquement pendant son fonctionnement.

Pour déployer une application, nous faisons un clic droit sur le serveur et choisissons l'item *Add and Remove* :



Dans l'écran qui apparaît, nous faisons passer un projet vers la droite pour le déployer, vers la gauche pour le sortir.
L'onglet *Console* permet de suivre le démarrage de Glassfish mais aussi de lire le journal de logs icône en forme d'écran pour alterner les vues).



Quand le projet a été déployé sans erreur, nous pouvons alors ouvrir notre navigateur préféré et lancer une requête sur le port 8080 vers la page *saisir.xhtml* (confer paragraphe au-dessus *Résultat attendu*) afin de faire fonctionner l'application.

Quand le projet ne fonctionne pas correctement, la console permet de voir des remontées d'erreur qui restent à analyser.
Cependant, quand nous sommes en mode développement, il est souvent utile de faire apparaître l'erreur sur la page web. Ceci se réalise en modifiant le fichier **web.xml**. Pour cela nous ajoutons dans ce fichier un nœud <context-param> dont la balise <param-value> a la valeur *Development*.

Nous pouvons aussi profiter de cette modification pour considérer les chaines vides des IHM en tant que *null*.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="http://java.sun.com/xml/ns/javaee/web
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/we
    version="3.0">
    <context-param>
        <param-name>javax.faces.PROJECT_STAGE</param-name>
        <param-value>Development</param-value>
    </context-param>
    <context-param>
        <param-name>javax.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL</param-name>
        <param-value>true</param-value>
    </context-param>
    <display-name>arboretum1</display-name>
    <servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>*.xhtml</url-pattern>
    </servlet-mapping>
</web-app>
```

Questionnement

Après avoir observé le fonctionnement de l'application ainsi que le code des différents éléments, vous répondrez aux questions suivantes en vous inspirant des renseignements ci-dessous éventuellement complétés par une recherche d'information appropriée :

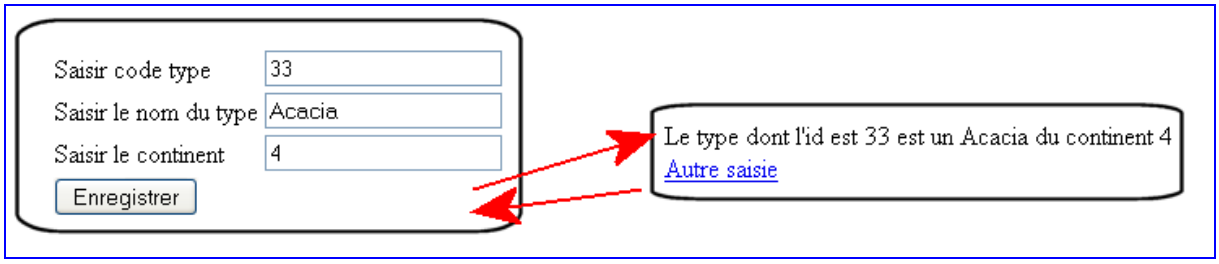
- un *ManagedBean* est une classe Java dont la fonction est de servir d'interface entre un fichier **.xhtml** et une classe métier. Ici, par exemple, entre TypeArbre et **saisir.xhtml** ;
- le *ManagedBean* proposé dispose d'une annotation @RequestScoped. Il existe aussi les annotations @ViewScoped et @SessionScoped ;
- les *Facelets* mettent en œuvre des composants correspondants à des balises XML dont la syntaxe et le sens rappellent le langage HTML. Par exemple <h :outputText>. La documentation proposée au début [4] permet d'approfondir leur rôle ;
- les *Facelets* échangent leurs données avec un *ManagedBean* par des EL (Expression Langage) dont la syntaxe est de la forme #{….} (confer document [1] Part II Chapter 6).

Questions	
1.A	D'après vous, qui a la responsabilité d'instancier la classe TypeArbreControleur ?
1.B	D'après vous, quel événement peut déclencher cette instanciation ?
1.C	Après avoir testé le fonctionnement de l'application en essayant successivement les trois annotations de portée @xxxScoped, déduisez-en la durée de vie d'un *ManagedBean* ainsi que le nombre d'instanciations réalisées dans chaque cas.
1.D	Vous observez l'EL présentée dans la fichier **saisir.xhtml** #{typeArbreControleur.typeArbre.id} Quelle est d'après vous la règle de nommage implicite des objets dans le framework JSF ?
1.E	Proposez un test permettant de valider ou non les affirmations suivantes : - une EL accède directement aux propriétés d'un objet ; - une EL accède aux propriétés d'un objet uniquement par l'intermédiaire des accesseurs.
1.F	Quel test pouvez-vous réaliser pour savoir si l'EL proposée au point 1.D correspond à une lecture, à une écriture ou à une lecture/écriture ?
1.G	Comment se réalise le passage de la vue **saisir.xhtml** à la vue **afficher.xhtml** et vice-versa ?
1.H	Le navigateur du client ne connaît que le langage HTML. En observant le code source de la page réceptionnée par votre navigateur à l'invocation de la page **saisir.xhtml**, expliquez la manière dont est traduite la balise <h:panelGrid> ?

Deuxième cas : contrôles des saisies

Nous ajoutons un identifiant de continent et allons tester les contrôles de saisie

Résultat à obtenir



Le bouton *Enregistrer* permet de faire apparaître les données saisies ; le lien *Autre saisie* permet de revenir au formulaire.
Nous allons apporter quelques modifications au projet précédent. La meilleure façon de procéder consiste à créer un nouveau projet pour conserver le premier projet en l'état.

Manipulation

Travail à faire	
2.1	Copiez le contenu de src du premier projet vers le second ainsi que les fichiers **.xhtml**
2.2	Modifiez la classe TypeArbre en y ajoutant une propriété correspondant au continent (confer copie ci-dessous)
2.3	Modifiez la classe TypeArbreControleur conformément à la copie d'écran ci-dessous.
2.4	Modifier la page **saisir.xhtml** selon la copie ci-dessous
2.5	Vous modifiez enfin la page **afficher.xhtml** afin que son rendu corresponde au besoin.

```
3  public class TypeArbre {
4      private Integer id;
5      private String nom;
6      //1..5 = afrique, amerique, asie, europe, oceanie
7      private Integer idcontinent;
8
9      public TypeArbre() {
```

```
@ManagedBean
@RequestScoped
public class TypeArbreControleur implements Serializable {
    private static final long serialVersionUID = 1L;
    private TypeArbre typeArbre;

    public TypeArbreControleur() {
        this.typeArbre = new TypeArbre();
        this.typeArbre.setIdcontinent(4); // europe par defaut
    }
    public TypeArbre getTypeArbre() {
        return typeArbre;
    }
    public String enregistre() {
        return "affiche";
    }
    public void verifieContinent(FacesContext context, UIComponent component,
            Object value) {
        Integer code = (Integer) value;
        if (code < 1 || code > 5) {
            ((UIInput) component).setValid(false);
            FacesMessage message = new FacesMessage("Continent inconnu");
            context.addMessage(component.getClientId(context), message);
        }
    }
}
```

```
<h:body>
    <h:form id="formSaisie">
        <h:panelGrid columns="3">
            <h:outputText value="Saisir code type" />

            <h:inputText id="id" value="#{typeArbreControleur.typeArbre.id}"
                required="true">
                <f:ajax event="blur" render="idMessage" />
            </h:inputText>
            <h:message id="idMessage" for="id" />

            <h:outputText value="Saisir le nom du type" />
            <h:inputText id="nom" value="#{typeArbreControleur.typeArbre.nom}"
                required="true">
                <f:ajax event="blur" render="nomMessage" />
            </h:inputText>
            <h:message id="nomMessage" for="nom" />

            <h:outputText value="Saisir le continent" />
            <h:inputText id="continent"
                value="#{typeArbreControleur.typeArbre.idcontinent}"
                validator="#{typeArbreControleur.verifieContinent}">
                <f:ajax event="blur" render="continentMessage" />
            </h:inputText>
            <h:message id="continentMessage" for="continent" />

            <h:commandButton action="#{typeArbreControleur.enregistre}"
                value="Enregistrer" />
        </h:panelGrid>
    </h:form>
</h:body>
```

Corps du fichier **saisie.xhtml**

Questionnement et modifications

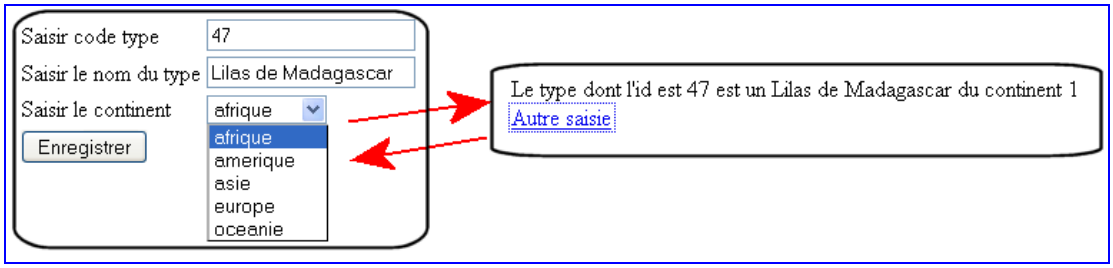
Après avoir déployé et testé l'application et analysé le code fourni, vous répondez aux questions suivantes et effectuez les modifications demandées.

Questions et modifications	
2.A	Par rapport à la première version, quel changement constatez-vous quant à la navigation de la page saisie vers la page affiche ?
2.B	Quel avantage obtient-on en déportant la responsabilité des liens vers le ManagedBean ? Modifiez le lien retour allant de afficher vers saisir.
2.C	Dans une EL, fait-on seulement appel à une propriété ?
2.D	Testez les différentes erreurs de saisie possibles : - id non numérique ; - champ nom laissé vide ; - n° de continent en dehors de la plage 1..5
2.E	Comment se comporte l'application lorsque le champ id est saisi avec une valeur non numérique et que le focus passe au champ nom ? Quel est le composant responsable de ce comportement dans la page **saisie.xhtml** ?
2.F	Pour chacune des trois catégories d'erreur de saisie proposée, quel objet a ici la responsabilité de sa gestion ? Qu'en pensez-vous ?
2.G	Les balises <h :message> permettent de relier un message d'erreur à un champ de saisie. Comment la méthode VerifieContinent () cible-t-elle le message à transmettre ?

Troisième cas : recours à une liste déroulante.

Saisir le numéro du continent n'est pas très significatif et c'est aussi source d'erreur. Nous allons mettre en place une liste déroulante de continents.
Remarque : pour rester dans le cadre d'une seule classe métier, nous utilisons pour l'instant une gestion des noms des continents « en dur ».

Résultat à obtenir



Manipulation

Travail à faire	
3.1	Après avoir créé un nouveau projet et récupéré les données du précédent, nous complétons la *ManagedBean* avec les données en dur ci-dessous (la classe TypeArbre est inchangée).
3.2	Modifier le body de la page **saisir.xhtml** selon la copie d'écran ci-dessous (l'autre page est inchangée)

```
private static Map<String, String> continents;
static{
    continents = new TreeMap<String, String>();
    continents.put("afrique","1");
    continents.put("amerique","2");
    continents.put("asie","3");
    continents.put("europe","4");
    continents.put("oceanie","5");
}
public Map<String, String> getContinents(){
    return continents;
}
```

Complément dans la classe du ManagedBean

```
<h:form id="formSaisie">
    <h:panelGrid columns="3">
        <h:outputText value="Saisir code type" />

        <h:inputText id="id" value="#{typeArbreControleur.typeArbre.id}"
            required="true">
            <f:ajax event="blur" render="idMessage" />
        </h:inputText>
        <h:message id="idMessage" for="id" />

        <h:outputText value="Saisir le nom du type" />
        <h:inputText id="nom" value="#{typeArbreControleur.typeArbre.nom}"
            required="true">
            <f:ajax event="blur" render="nomMessage" />
        </h:inputText>
        <h:message id="nomMessage" for="nom" />

        <h:outputText value="Saisir le continent" />
        <h:selectOneMenu value="#{typeArbreControleur.typeArbre.idcontinent}">
            <f:selectItems value="#{typeArbreControleur.continents}" />
        </h:selectOneMenu>
        <h:message />

        <h:commandButton action="#{typeArbreControleur.enregistre}"
            value="Enregistrer" />
    </h:panelGrid>
</h:form>
```

Questionnement et modifications

Après avoir déployé et testé l'application, mais aussi analysé le code fourni, vous répondez aux questions suivantes et effectuez les modifications demandées.

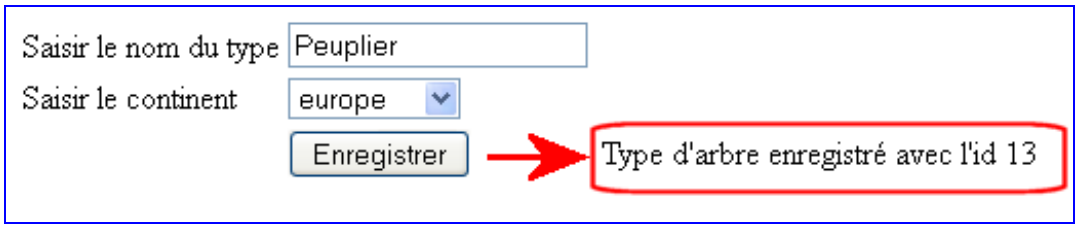
Questions et modifications	
3.A	Le framework impose-t-il de recourir à la technologie ajax pour tous les composants d'un FaceLet ?
3.B	L'EL #{typeArbreControleur.continents} reçoit une collection TreeMap. Qui a la responsabilité de l'itération sur ce Map pour remplir la liste déroulante ?
3.C	Après avoir observé le code source correspondant à la page **saisir.xhtml** reçue par le client, identifiez le rôle des composants de la Map.

Quatrième cas : gestion de la persistance.

Jusqu'à maintenant, la classe métier TypeArbre a bien été instanciée ou mise à jour à chaque saisie, mais son contenu n'est pas persistant c'est à dire disponible lors d'une prochaine connexion.
La gestion de la persistance à l'aide de l'API JPA va proposer une solution basée elle aussi sur l'emploi d'annotations.

Résultat à obtenir

L'interface est réduit : nous saisissons un type d'arbre et sélectionnons son continent dans la liste déroulante. Après avoir cliqué sur le bouton Enregistrer, un message nous confirme l'action en retournant la clé automatique gérée par MySql.



Création de la base de données

Nous choisissons ici, pour travailler dans un environnement de SGBD connu, de recourir à une base de données MySql plutôt qu'à la base Derby embarquée par Glassfish.

À l'aide d'un script, vous créez dans MySql une base de données « arboretum » contenant la table TYPEARBRE :

```
CREATE TABLE TYPEARBRE
 (
   ID INTEGER(2) PRIMARY KEY AUTO_INCREMENT ,
   NOM VARCHAR(32) ,
   IDCONTINENT SMALLINT(1)
 )
ENGINE=InnoDB;
```

Vous créez ensuite un utilisateur disposant des privilèges de lecture/écriture sur cette base de données, à défaut de tous les privilèges (éviter d'utiliser le compte root qui n'a pas vocation à être utilisé pour réaliser des traitements dans un contexte multiutilisateur).

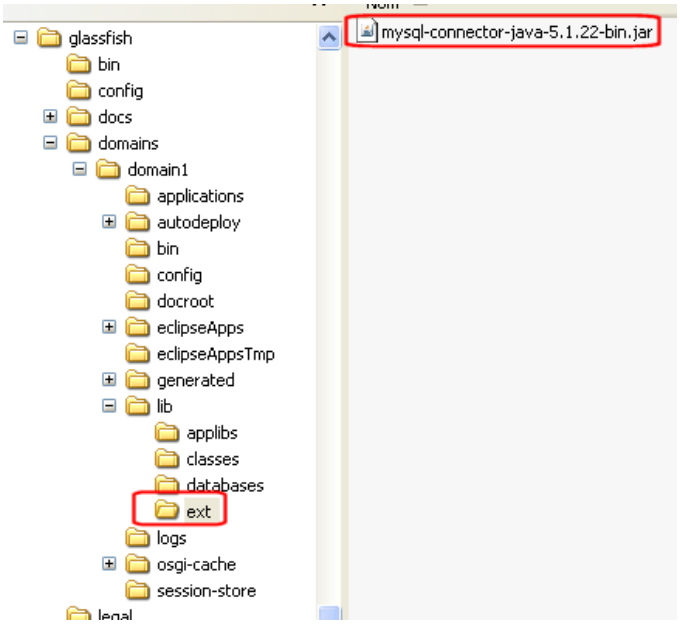
Vous peuplez ensuite la table avec quelques types d'arbres pour tester l'application.

Création d'un pool de connexion sur Glassfish

La connexion à la base de données peut se faire par une méthode classique : ouverture de connexion, action, fermeture.

Cette méthode trouve ses limites lors d'accès concurrents trop nombreux. La solution d'un pool de connexion s'avère plus performante et surtout plus rapide.

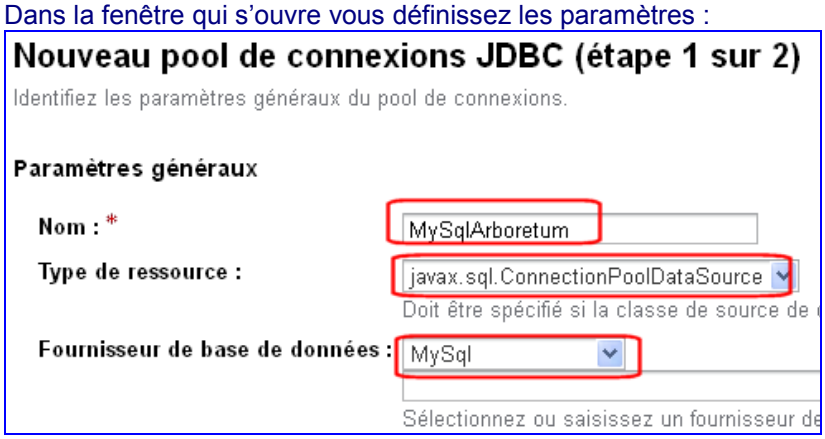
Pour pouvoir accéder à un SGBD autre que Derby, Glassfish doit disposer du connecteur MySql spécifique que vous téléchargez et copiez dans le répertoire *ext* montré ci-contre :



Vous ouvrez ensuite la console d'administration de Glassfish sur le port 4848 :
http://localhost:4848

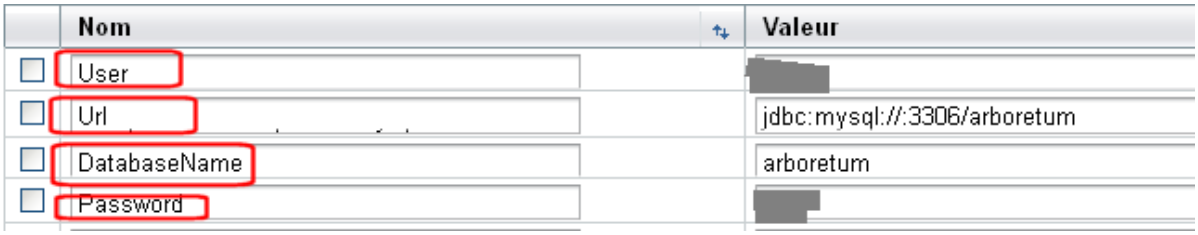
Quelques instants plus tard (soyez patient) vous pouvez accéder à la création d'un nouveau pool :

Dans la fenêtre qui s'ouvre vous définissez les paramètres :



Le nom du pool est arbitraire, ici nous choisissons MySqlArboretum.

Dans le formulaire suivant, parmi les 207 propriétés requises par le *driver MySql*, vous saisissez les quatre suivantes (en particulier l'Url qui n'est que partiellement remplie par défaut) :

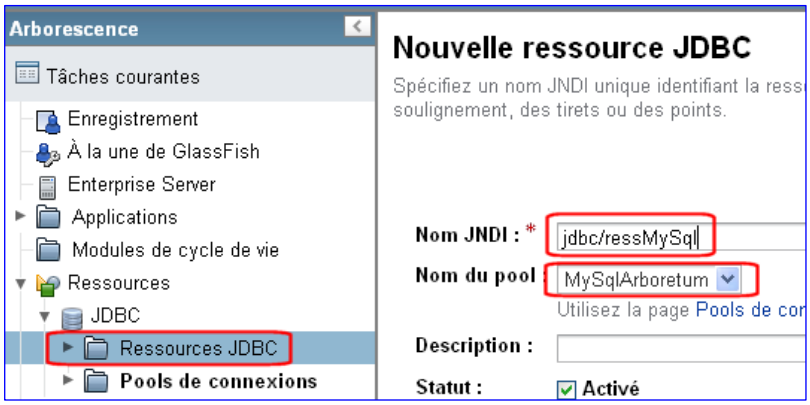


montage des propriétés utiles

Lorsque le paramétrage est terminé et si la base de données est accessible vous devez pouvoir réaliser un *ping* sur son port 3306 :

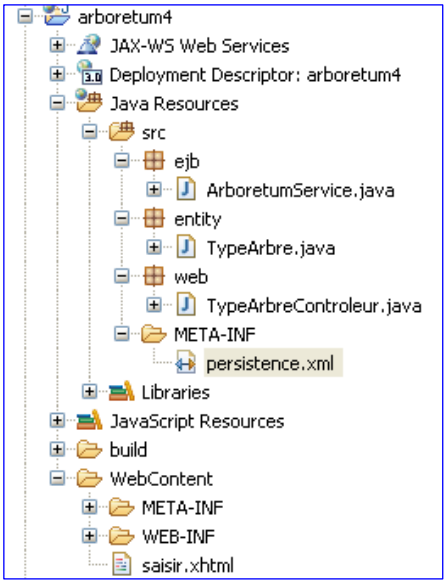
Création d'une ressource JDBC

La création d'une ressource JDBC basée sur le pool de connexion va permettre à l'API JPA d'accéder à la base de données. Vous lui donnez un nom arbitraire JNDI qui sera utilisé dans l'application java. Ici nous choisissons de la nommer *jdbc/ressMySql* :



Construction de l'application

Après avoir créé un nouveau projet et copié les classes et fichiers du précédent projet (sans **afficher.xhtml** qui n'est pas utile ici), vous devez arriver à l'organisation ci-contre :



Le fichier **persistence.xml** est à placer impérativement dans un répertoire META-INF situé dans *src*. C'est un fichier XML permettant l'accès à la base de données via la ressource JNDI dont le nom est fourni dans la balise <jta-data-source>. Le nom de l'unité de persistance attribué ici dans la balise <persistence-unit> sera utilisé par l'application.

Le contenu du fichier **persistence.xml** vous est fourni :

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
	xmlns="http://java.sun.com/xml/ns/persistence"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
		http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
	<persistence-unit name="arboretum" transaction-type="JTA">
		<jta-data-source>jdbc/ressMySql</jta-data-source>
		<properties>
			<property name="eclipselink.logging.level.sql" value="FINE" />
		</properties>
	</persistence-unit>
</persistence>
```

La balise <property name="eclipselink.logging.level.sql" value="FINE" /> va permettre de voir dans la console les requêtes SQL envoyées à la base de données par le framework lors des opérations d'accès à la base. Cette propriété sera bien évidemment à supprimer lors de la mise en production.

Manipulation

Travail à faire	
4.1	Après avoir créé un nouveau projet, récupéré les données du précédent et mis en place le fichier **persistence.xml**, vous ajoutez à la classe TypeArbre les trois annotations fournies ci-dessous.
4.2	Vous créez la classe ArboretumService montrée ci-dessous en veillant bien à ses annotations ; vous la placez dans le nouveau package *web* (pour bien organiser votre projet).
4.3	Vous modifiez la classe TypeArbreControleur en ajoutant et modifiant les données indiquées dans la copie d'écran ci-dessous.
4.4	Vous modifiez enfin le body de la page **saisir.xhtml** selon les indications ci-dessous.

```
8  @Entity
9  public class TypeArbre {
10     @Id
11     @GeneratedValue(strategy=GenerationType.IDENTITY)
12     private Integer id;
13
14     private String nom;
```

Les trois annotations à ajouter à la classe TypeArbre

```
10 @Local
11 @Stateless
12 public class ArboretumService {
13
14     @PersistenceContext(unitName="arboretum")
15     private EntityManager em;
16
17     public void creerTypeArbre(TypeArbre typeArbre){
18         em.persist(typeArbre);
19     }
20 }
```

La classe ArboretumService

```java
@ManagedBean
@RequestScoped
public class TypeArbreControleur implements Serializable {

    private static final long serialVersionUID = 1L;

    @EJB
    private ArboretumService arboretumService;

    private TypeArbre typeArbre;

    public TypeArbreControleur() {
        this.typeArbre = new TypeArbre();
        this.typeArbre.setIdcontinent(4); // europe par defaut
    }

    public void enregistre() {
        arboretumService.creerTypeArbre(typeArbre);
        FacesContext.getCurrentInstance().addMessage(null,
            new FacesMessage("Type d'arbre enregistré avec l'id " + typeArbre.getId()));
    }

    public TypeArbre getTypeArbre() {
```

Modifications dans la classe TypeArbreControleur

```xml
<h:body>
    <h:form id="formSaisie">
        <h:panelGrid columns="3">

            <h:outputText value="Saisir le nom du type" />
            <h:inputText id="nom" value="#{typeArbreControleur.typeArbre.nom}"
                required="true" />
            <h:message id="nomMessage" for="nom" />

            <h:outputText value="Saisir le continent" />
            <h:selectOneMenu value="#{typeArbreControleur.typeArbre.idcontinent}">
                <f:selectItems value="#{typeArbreControleur.continents}" />
            </h:selectOneMenu>
            <h:outputText />

            <h:outputText />
            <h:commandButton action="#{typeArbreControleur.enregistre}" value="Enregistrer" />
            <h:messages globalOnly="true" layout="table"/>
        </h:panelGrid>
    </h:form>
</h:body>
```

Le body de la page saisir.xhtml

Questionnement et modifications

Après avoir déployé et testé l'application puis analysé le code fourni, vous répondez aux questions suivantes et effectuez les modifications demandées.

Nous vous fournissons quelques indications que vous pourrez compléter par des recherches personnelles :

- l'annotation @Entity permet de rendre persistante la classe concernée dans une table de base de données ;
- un objet de la classe EntityManager permet de gérer les transactions et les accès à une base de données en lecture/écriture/modification/suppression ;
- l'annotation @EJB correspond à l'injection de dépendance : la classe cible est injectée par le framework dans la classe hôte et ses services sont alors disponibles.

Questions et modifications	
4.A	Comment se font, d'après vous, les mises en relation : - entre le nom de la classe et le nom de la table ? - entre les noms des propriétés et les noms de champs ?
4.B	Recherchez, par exemple dans la documentation [3], l'annotation spécifique permettant de mettre explicitement en correspondance une propriété et un champ
4.C	Qui a, d'après vous, la responsabilité de l'instanciation de l'objet *em* de la classe *EntityManager* ?
4.D	À l'aide d'un schéma représentant chaque élément sous forme d'un carré, allant de l'élément « pool de connexion » à l'élément « classe java utilisant ce pool », représentez le flux correspondant à une persistance.
4.E	Recherchez dans la documentation le rôle et l'utilisation de la balise <h:messages globalOnly="true" layout="table"/> utilisée dans la page **saisir.xhtml**
4.F	Les zones de saisie ne sont pas réinitialisées après chaque enregistrement. Modifiez l'application pour simplifier cette saisie.
4.G	Lors d'un enregistrement, la console montre les requêtes SQL envoyées à MySql : - INSERT INTO …. - SELECT …. Quel est le rôle de chacune d'elle ? Que pensez-vous de l'ordre dans lequel elles se présentent ?

Cinquième cas : affichage des données enregistrées

Nous souhaitons maintenant présenter les données enregistrées dans la base de données.

Résultat à obtenir

L'application ne va maintenant proposer qu'une seule page web **afficher.xhtml** qui va montrer le contenu de la table mise à jour précédemment.

Les continents ne sont montrés que sous la forme de leur identifiant.

1	Acacia	1
2	Arbousier	4
3	Cannelier	3
4	Caroubier	4
5	Cerisier	4
6	Cognassier	4
7	Cornouiller	4
8	Eucalyptus	5
9	Frene	4

Manipulation

La classe TypeArbre reste à l'identique ; le fichier **persistence.xml** aussi ; la page **saisie.xhtml** n'est pas nécessaire ici.

Travail à faire	
5.1	Après avoir créé un nouveau projet, récupéré les données utiles du précédent, vous ajoutez à l'EJB ArboretumService la méthode montrée ci-dessous.
5.2	Vous complétez le *ManagedBean* avec les propriété et méthodes fournies.
5.3	Vous créez enfin une *Facelet* **afficher.xhtml** dont le body sera alimenté avec le code ci-dessous.

```
public List<TypeArbre> donneListeTypeArbre(){
     Query query = em.createQuery("Select t FROM TypeArbre t");
     List<TypeArbre> res = query.getResultList();
     return res;
}
```

Méthode à ajouter dans la classe ArboretumService (l'EJB)

```
    private List<TypeArbre> listeTypeArbre;

    @PostConstruct
    public void init(){
         this.listeTypeArbre = arboretumService.donneListeTypeArbre();
    }
    public List<TypeArbre> getListeTypeArbre(){
         return listeTypeArbre ;
    }
```

Propriété et méthodes à ajouter dans la classe TypeArbreControleur (le *ManagedBean*)

```
<h:body>
    <h:dataTable value="#{typeArbreControleur.listeTypeArbre}" var="ligne" border="1">
        <h:column>
            <h:outputText value="#{ligne.id}" />
        </h:column>
        <h:column>
            <h:outputText value="#{ligne.nom}" />
        </h:column>
        <h:column>
            <h:outputText value="#{ligne.idcontinent}" />
        </h:column>
    </h:dataTable>
</h:body>
```

Contenu du body à placer dans le fichier **afficher.xhtml**

Questionnement et modifications

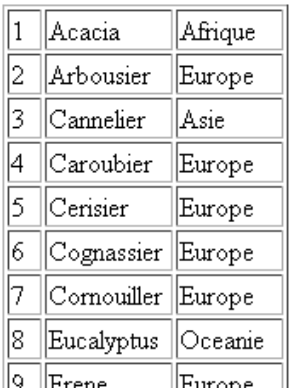
Après avoir déployé et testé l'application puis avoir analysé le code fourni, vous répondez aux questions suivantes et effectuez les modifications demandées.

Questions et modifications	
5.A	Quelle est la requête SQL envoyée à MySql lors de l'exécution de l'application ?
5.B	La méthode donneListeTypeArbre() de la classe ArboretumService montre une requête JPQL (confer document [2]). - sur quel type de donnée porte la clause FROM ? - lors de son exécution, quels objets sont instanciés ?
5.C	La méthode init() de la classe TypeArbreControleur (le *ManagedBean*) supporte l'annotation @PostConstruct. D'après vous quel est le rôle de cette annotation ? Aurait-on pu s'en dispenser ?
5.D	L'EL #{typeArbreControleur.listeTypeArbre} récupère une collection List. - qui a la responsabilité de son parcours lors de l'affichage des données ? - quelle structure de traitement vous semble mise en œuvre lors de l'utilisation de la balise <h:dataTable> ?
5.E	Modifiez l'application pour : - ne présenter que les types d'arbre du continent n°4 - les afficher par ordre alphabétique croissant.

Sixième cas : mise en base des continents

Les libellés des continents ont été placés jusqu'ici dans une collection statique. Nous pouvons admettre que leur libellé soit effectivement suffisamment constant pour justifier cette pratique. Cependant le rapprochement (plus précisément une jointure en langage base de données) avec les types d'arbres n'est pas rendu très simple avec cette méthode.
Nous allons donc construire une table CONTINENT dans la base de données.

Résultat à obtenir



1	Acacia	Afrique
2	Arbousier	Europe
3	Cannelier	Asie
4	Caroubier	Europe
5	Cerisier	Europe
6	Cognassier	Europe
7	Cornouiller	Europe
8	Eucalyptus	Oceanie
9	Frene	Europe

L'application est très semblable à la précédente : elle consiste seulement à améliorer l'affichage par la présentation du libellé du continent.

Modification de la base de données

De préférence à l'aide d'un script, vous modifiez la structure et le contenu de la base de données :

- ajout d'une table CONTINENT ;
- clé étrangère entre TYPEARBRE et CONTINENT ;
- peuplement de la table CONTINENT.

Manipulation

Le *ManagedBean* et l'EJB restent à l'identique, le fichier **persistence.xml** aussi.

Travail à faire	
6.1	Après avoir créé un nouveau projet, récupéré les données utiles du précédent, vous créez une classe Continent dans le package *entity*, en prenant éventuellement modèle sur la classe TypeArbre pour les annotations.
6.2	Vous complétez la classe TypeArbre en y ajoutant la propriété **continent** précédée des annotations fournies ci-dessous.
6.3	Vous modifiez la page **afficher.xhtml** pour qu'elle produise le résultat attendu en utilisant l'EL "#{ligne.continent.nom}"

```
@ManyToOne
@JoinColumn(name="IDCONTINENT")
private Continent continent;
```

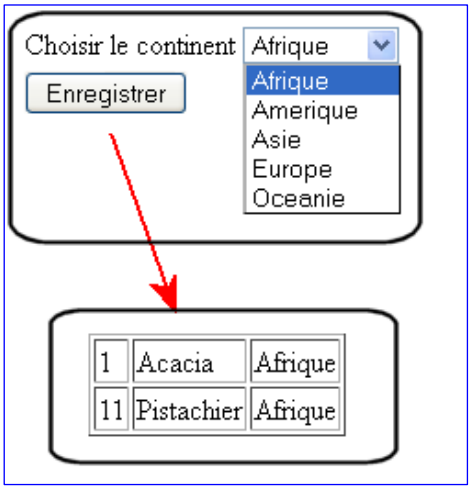
Propriété à ajouter dans la classe TypeArbre

Questionnement et modifications

Après avoir déployé et testé l'application et enfin analysé le code fourni, vous répondez aux questions suivantes et effectuez les modifications demandées.

Questions et modifications	
6.A	L'annotation @ManyToOne matérialise le lien entre TypeArbre et Continent. Dans le cas présent, s'agit-il d'un lien unidirectionnel ou d'un lien bidirectionnel ?
6.B	En observant la console, quelles sont les requêtes SQL envoyée à MySql lors de l'exécution de l'application ?
6.C	La requête JPQL présente dans l'EJB reste inchangée. Dans un environnement relationnel, on aurait pensé réaliser une requête de jointure. Comment pouvez-vous expliquer le résultat obtenu cependant ?
6.D	Dans le résultat attendu ci-dessus, l'arbre n° 2 et l'arbre n° 4 proviennent du même continent. En analysant les requêtes SQL, expliquez comment se comporte le framework à cet égard.
6.E	En démarrant dans le système d'exploitation le service *mysqld* afin qu'il mémorise dans un fichier de logs les véritables requêtes SQL reçues par le serveur MySql : - vérifiez la réalité du requêtage montré en 6.B ; - observez les *threads* de connexion et comparez les ouvertures ainsi réalisées au pool de connexion créé précédemment (nombre de connexions simultanées).

Septième cas : liaison bidirectionnelle



Résultat à obtenir

L'application évolue avec la présence de deux pages :
- **saisie.xhtml** permet de sélectionner un continent dans une liste déroulante ;
- le bouton *Enregistrer* permet alors de faire apparaître la page **afficher.xhtml** montrant les arbres provenant de ce continent.

Modifications à concevoir

Après avoir créé un nouveau projet, vous allez construire l'application en vous inspirant des cas précédents.

Questions et modifications	
7.A	Gestion des modèles En recherchant à l'aide par exemple du document [3] les annotations permettant de faire du *mapping* entre classes, vous créez un lien de Continent vers TypeArbre.
7.B	Gestion du bean contrôleur Vous modifiez le *ManagedBean* pour servir les interfaces.
7.C	Gestion de la classe de services Vous modifiez l'EJB afin qu'il fournisse les services attendus.
7.D	Gestion des vues Vous construisez les deux vues nécessaires.

Synthèse

Vous concevez un schéma montrant une représentation MVC incluant la classe FacesServlet ainsi que les classes construites à l'étape 7 précédente.