ISMA 3EME ANNEE

Compte Rendu d'intégration d'application

Compte Rendu Final

Maxime ESCOURBIAC Jean-Christophe SEPTIER

19/12/2011

Table des matières

Table des figures:

Introduction

Pour ce tp nous avons réalisé une entreprise application JEE dans son ensemble. Cette application a pour objectif d'utiliser une base de données sur les prises de rendez-vous d'un médecin. L'application en elle-même n'est pas complète, puisque l'objectif était d'apprendre les notions entourant les technologies JEE, et non de réaliser une application opérationnelle.

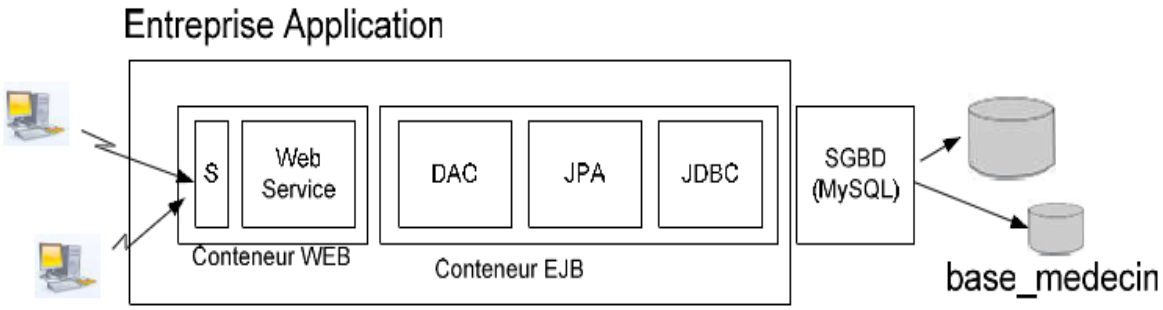
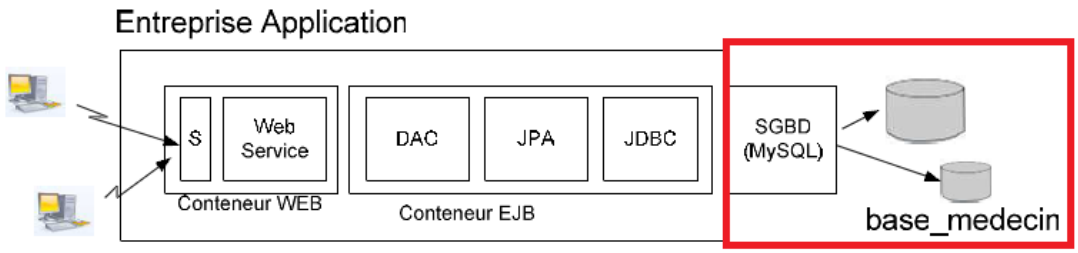


Figure 1: Schéma d'une Entreprise Application

Ce schéma représente l'ensemble des composants permettant de réaliser l'Entreprise Application. Nous avons réalisé l'ensemble des composants. Nous décrirons dans ce rapport leurs rôles, leurs fonctionnements, et de quelle façon nous les avons créés.

1. Le SGBD:



1.1. Définitions :

Une base donnée permet de stocker physiquement un ensemble d'information structuré selon un modèle. Pour notre application, la base de données a été créée sur notre PC de développement. Nous avons utilisé pour notre application MySql. MySQL est un système de bases de données relationnelles. Il est multi-utilisateur. Il permet d'administrer des bases de données, destinées à stocker et partager des informations en garantissant leur qualité, pérennité, et leur sécurité. MySql est ici un bon choix puisqu'il a été développé pour être très performant en lecture. Hors l'application devra plus souvent afficher des informations de la base que les mettre à jour.

MySql possède une licence libre pour une utilisation dans un logiciel libre, ou propriétaire sinon. Les autres principaux serveurs de base de données présents sur le marché sont Oracle et IBM DB2.

1.2. Modèle utilisé :

Pour l'application, nous avons utilisé le modèle suivant :

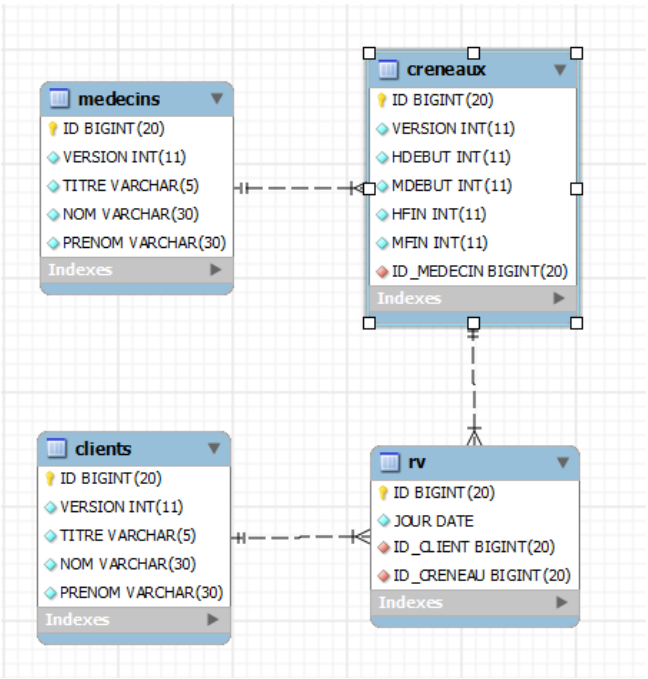


Figure 2: Diagramme de classe obtenu par WorkBench

Un médecin possède un titre, un nom et un prénom. Il en est de même pour les clients. Un créneau possède une heure de début, une heure de fin, et est relié à un médecin. Un rendez-vous possède un jour, mais aussi un client et un créneau.

1.3. Création de la base :

La base a été créée grâce au logiciel WorkBench. C'est un outil graphique de base de données. Il permet de manière simple de créer, d'administrer et de maintenir une base de données MySql. Nous avons tout d'abord créé manuellement les tables de la base de données, afin de se familiariser avec le logiciel. Nous créons d'abord la base, puis un schéma nommé base_medecin. Tout se fait de manière très simple et graphiquement. Nous choisissons le port, un utilisateur et le mot de passe de celui-ci. L'adresse est l'adresse locale (Localhost).

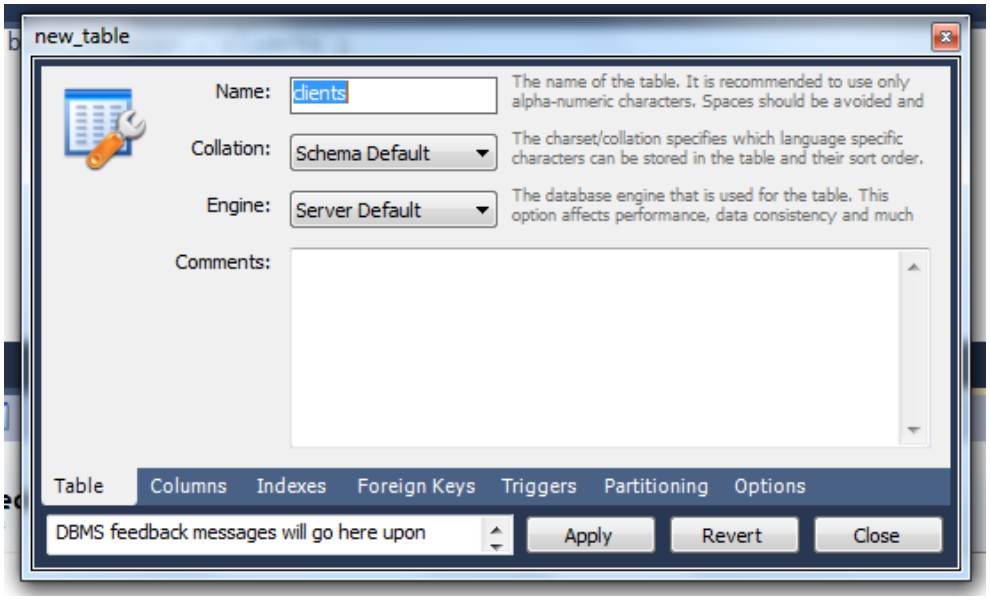
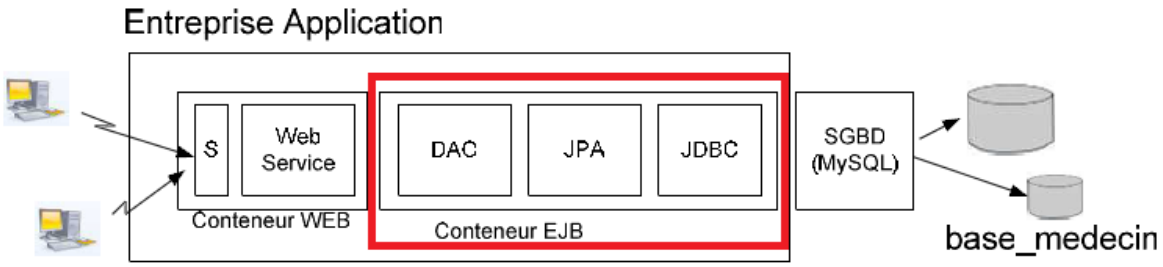


Figure 3: Fenêtre de création d'une table

Par exemple, cette fenêtre permet de créer une table, avec les champs de la tables, les indexes pour rendre plus rapide la recherche d'information, la clé étrangère pour relier les tables entre elle, mais également des triggers si on le souhaite.

Afin d'avoir une base cohérente avec le reste de la classe et la remplir, nous avons finalement recréé cette base de donnée à l'aide d'un script SQL. Nous avons alors à notre disposition une base de donnée remplies de certains champs sur un port de notre machine (ici le port 3306).

2. Le conteneur EJB :



1.1. Définition :

L'EJB (Entreprise Java Bean) est un composant serveur pour les plateformes JEE. Il permet d'implémenter la logique métier de notre application. Il est accessible par les clients et s'exécute dans un conteneur EJB. Il permet de réaliser facilement une application multicouche. Le client permettra d'afficher la présentation, et l'EJB servira de logique métier, grâce à la proposition de service. Le serveur prend en charge la création et la destruction des EJB.

Depuis la version 3.0, les fichiers de configurations ne sont plus obligatoires, et les annotations java permettent de spécifier entièrement la configuration et les propriétés de l'EJB.

Il existe deux modes de gestion d'EJB :

- Stateless : L'état de l'EJB n'est pas conservé entre 2 appels. Il est partagé par plusieurs clients. Il supporte donc les montées en charge.
- Stateful : L'état est conservé pendant la session Bean-client. Il n'y a donc qu'un seul client à la fois.

Le choix du mode dépend de l'application (nombre d'utilisateur, besoin des données pendant l'ensemble de la transaction…). Dans notre cas, on utilisera un mode Stateless. En effet :

- Un grand nombre d'utilisateur peut utiliser ces services en même temps. Il faut donc avoir une bonne montée en charge.
- Nous n'avons pas besoin de garder les informations du client pendant l'ensemble de la transaction.

Le client va rechercher l'EJB grâce à son nom logique JNDI. JNDI (Java Naming and Directory Interface) est une API Java de connexions à des annuaires qui permet de lier un nom à une information.

1.2. L'IDE NetBeans et Glassfish :

NetBeans est un EDI (*Integrated Development Environment*). C'est un logiciel regroupant un ensemble d'outils pour le développement de logiciels. On y trouve un debugger, un accès et une gestion de bases de données, serveurs Web, ressources partagées. Cet IDE a été choisi car il permet de faciliter la création de l'EJB.

L'avantage des IDE est de faciliter grandement l'écriture complète des programmes informatiques. Mais ils possèdent en général un très grand nombre de fonction que l'on n'utilise pas forcement, et qui peut alourdir le programme créé.

Nous allons également utiliser GlassFish, qui est un serveur d'application JEE. Il permet d'exécuter le code Java que nous allons écrire. GlassFish est intégré à NetBeans.

Nous pouvons donc créer un nouveau projet dans NetBeans de type « EJB Module», afin de créer notre conteneur EJB.

1.3. Le JDBC :

JDBC (*Java DataBase Connectivity*) est une interface de programmation qui permet à une application d'accéder à une base de données. Dans notre cas, il va permettre de connecter notre conteneur EJB à notre base de données. Après avoir connecté NetBeans à la base de données. La création du JDBC est complétement automatisé.

On créé une DataSource en ajoutant un « JDBC ressource ». Une DataSource est une interface représentant une fabrique de connexions vers la source de données physique. Cette DataSource s'implémente sous forme d'un pool de connexions. C'est un mécanisme permettant de réutiliser les connexions créées. Ceci est beaucoup moins lourd que la création systématique d'une nouvelle connexion.

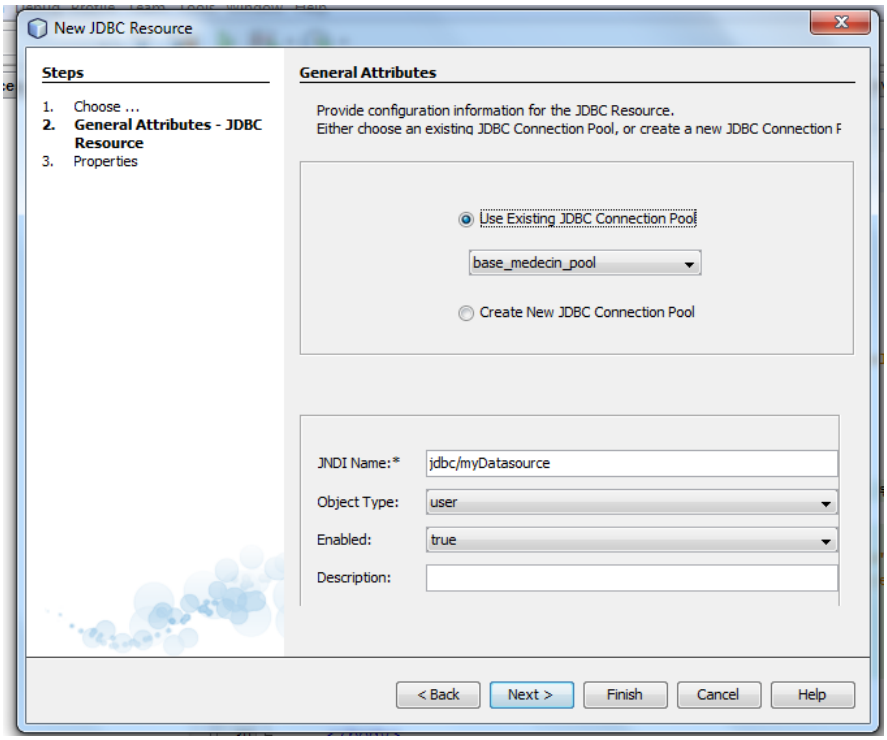


Figure 4: Fenêtre de création du pool de connexion

1.4. La JPA :

1.4.1. Définition :

La Java Persistence API (abrégée en JPA), est une interface de programmation Java permettant aux développeurs de créer un pont entre le monde relationnel de la base de données et le monde objet manipulé par le langage Java. Le pont peut se faire grâce à des fichiers Xml, ou, depuis le JDK 1.5, par des annotations java. Le JPA permet donc un mapping d'un objet avec une ou plusieurs tables de la base de données.

Elle permet également de manipuler des entités pour réaliser les opérations de persistance comme la récupération et la mise à jour d'objet. Les entités sont des classes qui représentent un ensemble de données de la base et récupérées sous la forme d'un tout.

JPA fournit également un langage de requête standard pour la récupération d'objet : le JP QL. Inspiré du SQL, il permet de réaliser des requêtes pour les entités sur les bases de données.

Il existe 3 types de relations entre entité :

- le one-to-one : une entité est liée à une et une seule entité.
- le many-to-one : plusieurs entités peuvent être reliées à une autre.
- le many-to-many : plusieurs entités peuvent être reliées à plusieurs autres.

Le mapping doit pouvoir gérer ces types de relations.

1.4.2. Création de la couche JPA :

Tout d'abord, il faut créer la JPA. Elle se fait automatiquement en créant une nouvelle « Persistance Unit ». On choisit comme fournisseur de persistance « Eclipse Link ». Eclipse Link est un framework de mapping objet-relationnel qui créé des correspondances entre notre base de données et les objets java. Les autres outils de mapping utilisés couramment sont Hibernate et TopLink.

On utilise également les transactions Java API. C'est une interface Java standard entre un gestionnaire de transaction et les différentes parties impliquées dans un système de transactions distribuées : le gestionnaire de ressources, le serveur d'application et les applications transactionnelles.

Une fois terminé, un fichier «persistence.xml» est créé. Celui-ci décrit l'ensemble des propriétés de notre JPA, comme la base utilisée ou le type de transaction.

Après la création de la JPA, on crée les entités JPA. Pour cela, on utilise l'interface utilisateur de NetBeans. On choisit « Classe d'entité de la base de données », puis on choisit les classes d'entités que l'on veut générer en fonction de la base choisie.

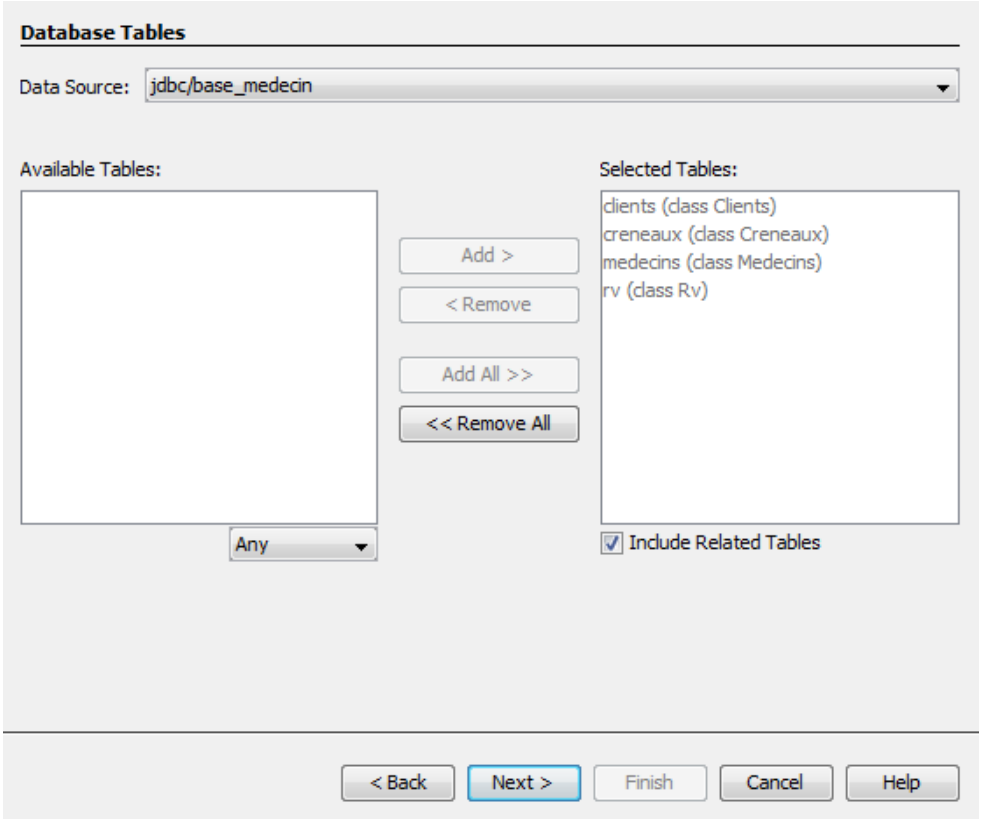


Figure 5: Création des entités JPA

Une fois choisie, on crée un nouveau package « JPA », puis on décide de générer les requêtes nommées pour les champs persistants. On a donc après ces étapes la création de 4 classes : Clients.java, Creneaux .java, Medecin.java et RV.java. Une grande partie du code a été générée, mais il faut toutefois compléter la classe RV car le constructeur avec l'id du médecin et l'id du créneau n'a pas été construit.

```
public Rv(Long id, Date jour, Clients idClient, Creneaux idCreneau ) {
    this.id = id;
    this.jour = jour;
    this.idClient = idClient;
    this.idCreneau = idCreneau;
}
```

Figure 6: Constructeur de RV

Voici ci-dessus un exemple de code généré.

```java
@Entity
@Table(name = "clients")
@XmlRootElement
@NamedQueries({
    @NamedQuery(name = "Clients.findAll", query = "SELECT c FROM Clients c"),
    @NamedQuery(name = "Clients.findById", query = "SELECT c FROM Clients c WHERE c.id = :id"),
    @NamedQuery(name = "Clients.findByVersion", query = "SELECT c FROM Clients c WHERE c.version = :version"),
    @NamedQuery(name = "Clients.findByTitre", query = "SELECT c FROM Clients c WHERE c.titre = :titre"),
    @NamedQuery(name = "Clients.findByNom", query = "SELECT c FROM Clients c WHERE c.nom = :nom"),
    @NamedQuery(name = "Clients.findByPrenom", query = "SELECT c FROM Clients c WHERE c.prenom = :prenom")})
public class Clients implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)
    @NotNull
    @Column(name = "ID")
    private Long id;
    @Basic(optional = false)
    @NotNull
    @Column(name = "VERSION")
    private int version;
    @Basic(optional = false)
    @NotNull
    @Size(min = 1, max = 5)
    @Column(name = "TITRE")
    private String titre;
    @Basic(optional = false)
    @NotNull
    @Size(min = 1, max = 30)
    @Column(name = "NOM")
    private String nom;
    @Basic(optional = false)
    @NotNull
    @Size(min = 1, max = 30)
    @Column(name = "PRENOM")
    private String prenom;
    @OneToMany(cascade = CascadeType.ALL, mappedBy = "idClient")
    private List<Rv> rvList;
```

Figure 7: Code de l'entité Clients

Ceci représente un morceau de la classe Clients. Des annotations java permettent de réaliser le mapping avec la base de données :
@Entity : Permet de définir la classe comme entité.

@Table : Désigne la table correspondante à cette entité.

@nameQueries : Correspond aux requêtes nommées.

@id : Indique quel champ de la classe est la clé primaire.

@GenerateValue : Permet de générer la clé selon une stratégie définie.

@NotNull : Le champ ne peut pas être nul.

@Column : Permet de lier le champ à une colonne de la table.

@Size : Définie la taille du champ.

@one-to-many : Définit quel type de relation il existe entre deux entités. Par exemple, dans ce cas, un client peut être lié à un rendez-vous (et un rendez-vous ne peut avoir qu'un seul client). On a donc une relation de type on-to-many lié sur l'idClient.

@basic : Définie la stratégie de récupération des données. La stratégie EAGER permet de charger les données directement alors que le Lazy loading consiste à charger des objets uniquement lorsque l'on y accède explicitement. Pour les collections (relations one-to-many et many-to-many) la valeur par défaut est LAZY mais pour les objets contenus dans un autre (relations many-to-one) la valeur par défaut est EAGER ce qui signifie que tout sous-objet est donc chargé.

1.4.3. Avantages et inconvénients :

JPA permet de réaliser plus facilement une couche. Les requêtes sont plus simples, et les erreurs remontées en cas de modification de la base sont plus simple à gérer. Elle donne également une grande stabilité et une standardisation, puisqu'il n'est pas nécessaire de changer de DAO en fonction de l'outil de mapping utilisé.

L'inconvénient de JPA est qu'elle est plus lourde que l'utilisation de JDBC uniquement. JPA étant de plus haut niveau que JDBC, les performances seront donc réduites.

Le choix ou non de son utilisation doit être fait selon l'utilisation de la base de données. Pour une utilisation intensive de la base de données, JDBC doit être utilisé, sinon on utilise JPA pour le gain de temps lors du développement.

1.5. Le DAO :

1.5.1. Définition :

Le DAO est une couche supplémentaire correspondant à la couche métier. Il permet de séparer la manipulation des objets dans l'application et les échanges avec la base de données comme la récupération de données ou la mise à jour.

Le DAO va utiliser un Entity Manager. Celui-ci est une interface qui permet de communiquer avec la persistance. On peut donc utiliser un ensemble de fonction que l'on peut utiliser dans le code métier pour interagir avec notre JPA.

1.5.2. Mise en place du DAO :

On va d'abord créer le « Session Bean » pour les classes d'entités grâce à l'interface. On choisit ensuite les classes d'entités JPA que l'on veut utiliser avec notre DAO.

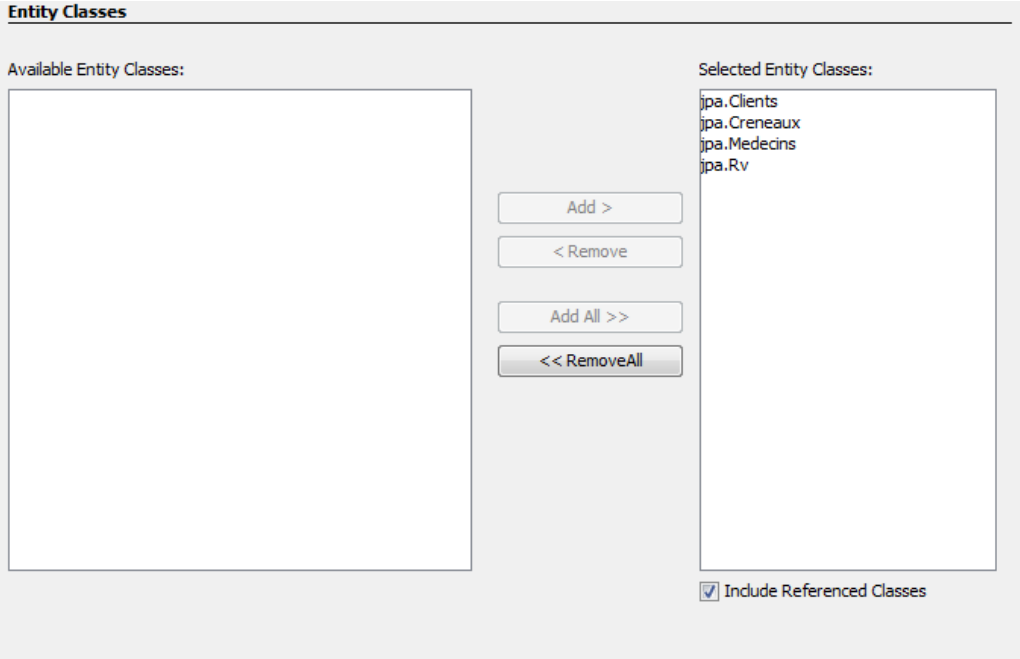


Figure 8: Fenêtre de création des façades Bean

Une fois les classes choisies, on entre le nom de notre package, puis on choisit de créer l'interface Local. On a donc la création de 7 classes.

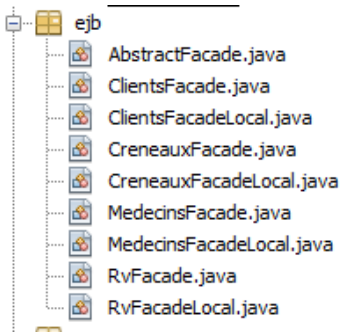


Figure 9: Classes créés

La classe AbstractFacade possède un ensemble de méthode commune à toutes les classes du DAO. C'est une méthode générique avec les comportements semblables des objets, comme la recherche par rang, la création, la mise à jour, la recherche ou le nombre de résultats.Il y a ensuite pour chaque objet une classe « facade » qui dérive d'AbstractFacade et qui implémente une interface « FacadeLocal ». Par exemple, pour l'entité Client, on a une classe ClientFacade, et une interface ClientFacadeLocal. Les façades utilisent un entity Manager.

On crée ensuite l'EJB session. C'est un objet proposant des services permettant d'utiliser les objets crées précédemment. Avec l'interface utilisateur, on choisit le nom de l'EJB. On choisit s'il est Stateless ou Stateful. La particularité principale d'un Stateful Session Bean est de conserver son état entre différents appels de méthodes, contrairement au Stateless Session Bean qui ne le conserve pas.

On choisit pour ce tp un Stateless pour éviter que le serveur conserve un état pour chaque connexion, c'est qui peut être lourd en cas de connexions nombreuses. On a donc créé une classe

« DaoJpa» qui implémente une interface « DaoJpaLocal » qui va permettre de réaliser des méthodes plus complètes comme getRvByAll() ;

Il existe plusieurs façon de réaliser la méthode getRvbyAll(). La première consiste à réaliser une requête nommée qui sera utilisée par le JPA grâce à l'Entity Manager (ici em).

```
String q = "SELECT e FROM RV as e"
        + "WHERE e.jour = :date";
Query query = em.createQuery(q);
query.setParamater("date",date);
List<Rv> listeRv = (List<Rv>) query.getResultList();

return listeRv;
```

Figure 10: Requêtes qui donne les RV selon le jour

On peut aussi créer la requête dans les entités JPA, puis les appeler ensuite dans le DAO.

```
@NamedQueries({
    @NamedQuery(name = "Rv.findAll", query = "SELECT r FROM Rv r"),
    @NamedQuery(name = "Rv.findById", query = "SELECT r FROM Rv r WHERE r.id = :id"),
    @NamedQuery(name = "Rv.findByJour", query = "SELECT r FROM Rv r WHERE r.jour = :jour")
})
```

Figure 11: Requêtes nommées

Ces méthodes pourront ensuite être appelées par les WebServices utilisant notre EJB.

Pour notre projet, j'ai créé une Session Bean dans un package nommé Dao. Cette session Bean permettra d'effectuer l'ensemble des requêtes dont j'aurais besoin.

La création de cette session Bean se fait par l'interface graphique de Netbeans. On choisit d'ajouter un nouveau « Session Bean », en choisissant si la session doit être Stateless ou Statefull. Une fois créé, on a donc une classe Dao et son interface, DaoLocal. On complète donc cette classe afin d'ajouter la méthode getAllClients retournant tout les clients.

```
/**
 *
 * @author JC
 */
@Stateless(mappedName="Interface")
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public class Dao implements DaoLocal {
    @PersistenceContext(unitName ="EJBModule1PU")
    private EntityManager em;

        public List<Clients> getAllClients()
    {
        return em.createNamedQuery("Clients.findAll").getResultList();

    }
}
```

Figure 12: Méthode getAllClients du DAO

Tout d'abord, l'annotation @stateless permet de dire que la classe Dao est un bean de type Stateless.

@TransactionAttribute permet de configurer comment le container gère les transactions quand un client invoque une méthode. Le TransactionAttributeType.REQUIRED correspond à deux propriétés :

- Le bean est toujours dans une transaction.
- Si l'appelant est dans une transaction, alors le bean la rejoint, sinon, le container crée une nouvelle transaction.

L'EJB va utiliser un Entity Manager. Celui-ci est une interface qui permet de communiquer avec la persistance. On peut donc utiliser un ensemble de fonction que l'on peut utiliser dans le code métier pour interagir avec notre JPA. L'annotation @PersistenceContext permet de définir avec quelle persistance l'Entity Manager va interagir. EntityManager va, pour la méthode getAllClients, appeler les requêtes nommées Clients.findAll créé précédemment dans la JPA et retourner le résultat sous forme de liste pour pouvoir être traité par les clients qui utiliseront cette méthode.

1.5. Déploiement de l'EJB :

Après avoir créé notre DAO, on doit déployer notre EJB pour qu'il puisse être utilisé par un WebService. Tout d'abord, on ajoute les librairies permettant de faire fonctionner l'EJB, c'est-à-dire les librairies EclipseLink et MySQL.

On ajoute ensuite un fichier de configuration permettant de configurer le JNDI. Il contient 3 lignes :

```
java.naming.factory.initial = com.sun.entreprise.naming.SerialInitContextFactory
java.naming.factory.url.pkgs = com.sun.enterprise.naming
java.naming.factory.state = com.corba.ee.impl.presentation.rmi.JNDIStateFactoryImpl
```

Figure 13: Fichier de configuration

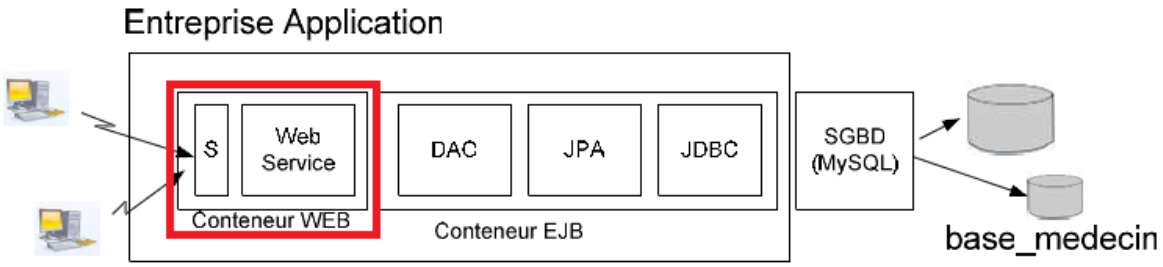
La valeur des paramètres correspondent à ceux d'un serveur GlassFish. Ils ont des valeurs différentes si on utilise un autre serveur.

Il permet de définir le contexte JNDI. Pour obtenir une instance de la classe InitialContext et ainsi réaliser la connexion au service, on a besoin de 3 paramètres :

- Java.naming.factory.initial : permet de préciser le nom de la fabrique proposée par le fournisseur. Cette fabrique est en charge de l'instanciation d'un objet de type InitialContext du fournisseur de service dans le contexte initial.
- Java.naming.factory.url.pkgs : définit le package de la fabrique.
- Java.naming.factory.state : Donne l'état initial de cette fabrique.

Une fois le fichier créé, on peut déployer l'EJB. Une fois déployé, on obtient un jar qui pourra être utilisé par un webService.

3. Création d'un WebService:



3.1. Définition :

Un WebService est un programme informatique permettant la communication et l'échange de données entre applications et systèmes dans des environnements distribués. Le WebService va fournir un ensemble de service à ces clients. Une fois déployé, le WebService possède une URL, et est accessible via les protocoles internet. Les réponses à une demande de service utilisent le protocole SOAP. C'est un protocole basé sur le XML, qui permet la transmission de message entre objets distants.

L'avantage principal des WebServices est de pouvoir être réutilisé par divers applications sans problème d'interopérabilité. L'inconvénient est que l'utilisation de WebServices est plus lourde que d'autre approche d'informatique répartie comme le CORBA.

3.2. Création d'un WebService et test :

On créé d'abord un nouveau projet WebApplication. Une fois créé, on ajoute le Jar créé lors de l'étape précédente afin de pouvoir utiliser les Sessions Beans créés précédemment.

On peut donc maintenant créé le WebService s'appuyant sur notre Bean créé dans l'EJB. On choisit le Bean « DAO ». Le WebService est ensuite créé automatiquement.

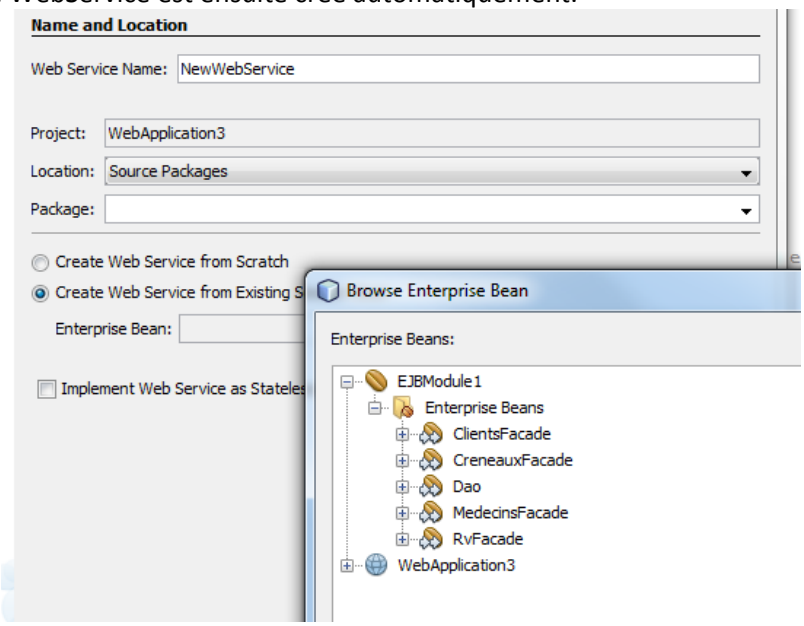


Figure 14: Choix du Dao

On a donc la création du WebService.

```
@WebService(serviceName = "NewWebService")
public class NewWebService {
    @EJB
    private DaoLocal ejbRef;// Add business logic below. (Right-click in editor and choose
    // "Insert Code > Add Web Service Operation")

    @WebMethod(operationName = "getAllClients")
    public List<Clients> getAllClients() {
        return ejbRef.getAllClients();
    }

}
```

Figure 15: WebService

L'annotation @WebService permet de définir que cette classe est un WebService, ainsi que son nom. @EJB permet de définir notre DaoLocal et l'EJB qui sera utilisé par le WebService. La méthode getAllClients sera une WebMethod, comme l'indique l'annotation. Celle-ci ne fait que renvoyer le contenu de la méthode getAllClients de l'EJB.

Une fois créé, on peut tester le webService en cliquant droit dessus et en choisissant « Tester le webService ». On ouvre donc notre navigateur Internet à l'adresse : http://localhost:8080/WebApplication3/WebService?Tester avec le port et l'adresse du serveur, la WebApplication et le WebService. Une page recense les méthodes et permet de les appeler.

Figure 16: Test du WebService

En cliquant sur getAllClients(), on affiche l'appel au WebService et la réponse.

SOAP Request

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Header/>
    <S:Body>
        <ns2:getAllClients xmlns:ns2="http://test/"/>
    </S:Body>
</S:Envelope>
```

Figure 17: Requète SOAP

L'appel se fait grâce au Protocol SOAP, qui permet la transmission de messages entre objets distants. On a donc l'adresse du package (ici « test ») et le nom de la méthode.

SOAP Response

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Body>
        <ns2:getAllClientsResponse xmlns:ns2="http://test/">
            <return>
                <id>1</id>
                <nom>MARTIN</nom>
                <prenom>Jules</prenom>
                <titre>Mr</titre>
                <version>1</version>
            </return>
            <return>
                <id>2</id>
                <nom>GERMAN</nom>
                <prenom>Christine</prenom>
                <titre>Mme</titre>
                <version>1</version>
            </return>
            <return>
                <id>3</id>
                <nom>JACQUARD</nom>
                <prenom>Jules</prenom>
                <titre>Mr</titre>
                <version>1</version>
            </return>
            <return>
                <id>4</id>
                <nom>BISTROU</nom>
                <prenom>Brigitte</prenom>
                <titre>Melle</titre>
                <version>1</version>
            </return>
        </ns2:getAllClientsResponse>
    </S:Body>
</S:Envelope>
```

Figure 18: Réponse SOAP

La réponse est également sous forme de SOAP. On a donc la liste des clients dans la base de données.

Nous avons, sur le même principe, créé d'autre WebMethode afin de retourner la liste de Rendez-Vous, des Médecins des créneaux, mais également de rechercher les informations selon l'id de l'objet.

```
public List<Clients> getClientById(int id)
{
    return em.createQuery("SELECT c FROM Clients c WHERE c.id =" + id).getResultList();
}
```

3.3. Création d'une page JSP :

Le JSP (JavaServer Page) permet de générer du code HTML grâce à du langage Java. On peut donc ajouter du code Java dans du code statique HTML. Les JSP sont compilés et transformé en Servlet Java, c'est-à-dire une classe Java qui permet de créer dynamiquement des données dans un serveur http.

Dans notre cas, cela correspond à la partie présentation. On va donc créer une page Web pour pouvoir accéder à quelques-unes des WebMethode créée précédemment.

Pour faciliter la tâche, notre client sera ajouté dans la même application que le WebService. Cette page JSP permet de découvrir de nouvelle notion. Aucune recherche esthétique n'a été faite.

3.3.1. Génération de code et référence :

On créé d'abord une référence au WebService que l'on a créé. Cela se fait automatiquement grâce à NetBeans en ajoutant un « Web Service Client ». On a donc une référence sur notre WebService que l'on rafraichit manuellement, pour pouvoir générer un code permettant d'utiliser notre WebService. Ce code est généré grâce à JAX-WS, qui est une API Java permettant de faciliter la création de WebService. Par exemple, une classe pour chaque objet a été créée, correspondant à celle de l'EJB. Une classe WSDaoJpa_Service permet d'accéder à notre WebService. Elle possède l'URL du WebService, et une méthode getWSDaoJpaPort() qui permet d'accéder au port.

Une fois ces classes générées, on créé d'abord une classe Java, appelée IndexClient. Cette classe va utiliser le WebService, et sera utilisé dans le JSP pour afficher les résultats.

```
public List<Clients> getClients()
{
    rendezvous_service.WSDaoJpa_Service ws = new WSDaoJpa_Service();

    List<Clients> myArr = new ArrayList<Clients>();
    myArr = ws.getWSDaoJpaPort().getAllClients();

    return myArr;
}
```

Figure 19: Méthode qui retourne les clients

On écrit une méthode getClients(), qui utilise le WSDaoJpa_Service, et va utiliser la WebMethod getAllClients(). On retourne donc une liste de client.

3.3.2. Page JSP :

On peut donc ajouter notre page JSP à notre projet afin d'afficher les résultats de l'appel au WebService. L'objectif est de créer des boutons pour afficher la liste des médecins, la liste des clients, et la liste des médecins par clients.

```
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Page</title>

    </head>
    <body>

         La date courante est : <%= new java.util.Date() %>

        <%
        String [] languages = { "Java", "C++", "Smalltalk", "Simula 67" };
        out.println("<h3>Principaux langages orientés objets : </h3>");
        for (int i = 0; i < languages.length; i++) {
            out.println("<p>" + languages[i] + "</p>");
        }
        %>


    </body>
</html>
```

Figure 20: Code d'exemple d'une JSP

Ce code permet de tester la page JSP. Le langage HTML est un langage de balise. Le corps de l'application « <body » permet de décrire ce qui sera afficher. Les balises « Head » permettent de délimiter des éléments qui seront valables pour l'ensemble de la page, comme le titre. On utilise les balises « <% » et « >% » afin de délimiter le code Java. Ici, on va donc générer du code, comme par exemple la méthode Date, qui permet d'afficher la date. La deuxième partie permet d'afficher le contenu d'une liste en le parcourant. C'est donc sur ce principe que devra être réalisé l'application.

La date courante est : Tue Dec 20 00:08:03 CET 2011

Principaux langages orientés objets :

Java

C++

Smalltalk

Simula 67

Figure 21: Résultat du code précédent

On va d'abord écrire le code HTML dans lequel sera intégré le code Java.

```
<body>
    <h1>Bienvenue</h1>

    <button onclick="getClients();">Liste des clients</button>
    <div id="clients"></div>

    <button onclick="getMedecins();">Liste des medecins</button>
    <div id="medecins"></div>

    <button onclick="getMedecinsClient();">Liste des medecins par clients</button>
    <div id="medecinsClients"></div>



</body>
```

Figure 22: Affichage des boutons

Dans le corps de la page HTML, on ajoute trois boutons, auxquels on fait correspondre une fonction JavaScript. Un clic sur ce bouton permet d'afficher nos résultats sans recharger la page Web.

```
<script type="text/javascript">

    function getClients() {
        <%
                beanJsp.IndexClient ind = new beanJsp.IndexClient();
                StringBuffer val = new StringBuffer();
                List<Clients> listClient = ind.getClients();


                for (Clients cl : listClient)
                  val.append(cl.getNom() + "<br />");
                %>
        var chaine = "<%= new String(val) %>";
        document.getElementById("clients").innerHTML = chaine;
    }



</script>
```

Figure 23: Fonction JavaScript

On ajoute donc ces fonctions JavaScript dans la balise Head de la page Web. Par exemple, la fonction ci-dessus permet d'afficher la liste des clients. On utilise donc la classe IndexClient afin de récupérer la liste des clients. On parcourt ensuite cette liste afin d'afficher le nom des Clients.

Figure 24: Affichages des noms de clients

On a donc bien l'affichage des clients sous le bouton la liste des clients après avoir cliqué sur « Liste des Clients ».

On fait donc de même pour pouvoir afficher la liste des médecins. Pour la liste des médecins par clients, il suffit de parcourir la liste des clients, puis de lister les médecins grâce à une méthode que nous avons écrite dans le DAO, et qui sera ajouter dans le WebService. Cette méthode réalise une requête en faisant une jointure sur les tables Médecins, Créneaux et RV.

Conclusion

Nous avons pu, grâce à ce TP, réaliser l'ensemble des composants d'une Entreprise Application, mais également un petit client de WebService avec une page JSP. Il nous a permis de rentrer dans les détails, afin de comprendre en profondeur les notions et le fonctionnement des conteneurs EJB, des WebServices, et du lien entre eux, bien que l'application produite ne soit pas utilisable.

Nous avons pu découvrir la puissance de l'IDE NetBeans, permettant de générer un grand nombre de fichier de configuration, afin de faciliter grandement la réalisation de cette Entreprise Application. En effet, bien que les relations entre chaque composant est complexe, mais cela se fait de manière aisée grâce aux différentes interfaces de NetBeans.

Nous avons donc maintenant la possibilité de réaliser des Entreprises Applications complètes utilisant les technologies JEE grâce aux connaissances acquises.