Construire un service web Java EE avec l'IDE Netbeans 6.5 et le serveur Java EE Glassfish

Serge Tahé, février 2009

http://tahe.developpez.com

Le texte qui suit fait référence aux documents suivants :

[ref1] : [http://tahe.ftp-developpez.com/fichiers-archive/javaee.pdf] - un cours Java EE
[ref2] : [http://tahe.ftp-developpez.com/fichiers-archive/jpa.pdf] - un cours JPA

Par facilité, ces références sont notées par la suite [**ref1**] et [**ref2**].

1 Objectifs et Outils

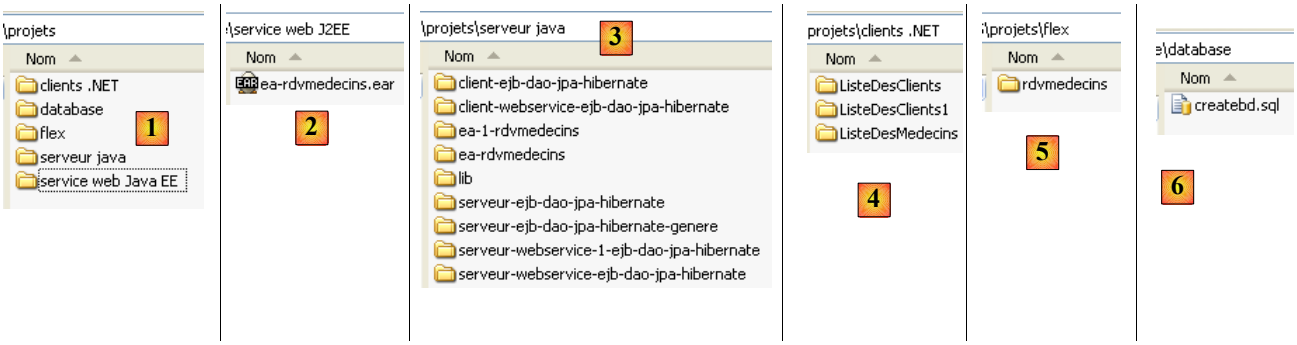
Le tutoriel vise à montrer les démarches qui mènent à la création et au déploiement d'un service web J2EE avec l'IDE Netbeans 6.5 et le serveur d'application Glassfish qui l'accompagne. Par ailleurs, nous présentons divers clients pour ce service web : clients Java, C#, Asp.Net, Flex.

Les outils utilisés sont :

- le SGBD MySQL 5 [http://www.mysql.com/]
- l'IDE Netbeans 6.5 [http://www.netbeans.org/]
- l'IDE Visual Studio Express 2008 SP1 (C# et Web) [http://www.microsoft.com/Express/]
- l'IDE Adobe Flex Builder 3 [https://www.adobe.com/cfusion/tdrc/index.cfm?loc=fr_fr&product=flex]
- le serveur Apache de l'outil Wamp [http://www.wampserver.com/]
- Adobe Flash Player : [http://www.adobe.com/fr/products/flashplayer/]

Le code n'est pas expliqué dans ses moindres détails. Aussi ce tutoriel est-il destiné à des personnes ayant une première expérience avec Netbeans, Java EE, Ejb3 et JPA. Tous les éléments utilisés dans ce tutoriel sont expliqués dans [**ref1**] et [**ref2**] mais pas nécessairement dans le tutoriel lui-même.

Le tutoriel est accompagné d'un fichier *zip* contenant les éléments suivants :



- [1] : le contenu du zip
- [2] : le service web JEE sous forme d'archive *ear*
- [3] : un ensemble de projets Netbeans 6.5 visant à construire progressivement le service web JEE
- [4] : des clients .NET du service JEE - un client C# et deux clients ASP.NET / C#
- [5] : les clients Flex 3 du service JEE
- [6] : le script de création de la base de données utilisée par l'application

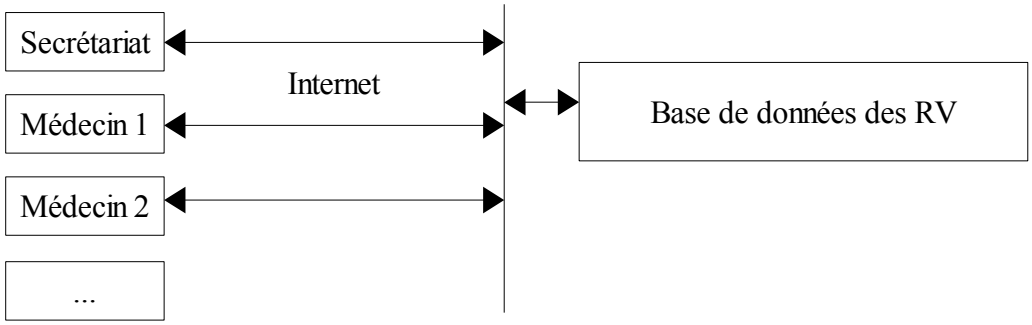
2 La nature du service web réalisé

Une société de services en informatique [ISTIA-IAIE] désire proposer un service de prise de rendez-vous. Le premier marché visé est celui des médecins travaillant seuls. Ceux-ci n'ont en général pas de secrétariat. Les clients désirant prendre rendez-vous téléphonent alors directement au médecin. Celui-ci est ainsi dérangé fréquemment au cours d'une journée ce qui diminue sa disponibilité à ses patients. La société [ISTIA-IAIE] souhaite leur proposer un service de prise de rendez-vous fonctionnant sur le principe suivant :

- un secrétariat assure les prises de RV pour un grand nombre de médecins. Ce secrétariat peut être réduit à une unique personne. Le salaire de celle-ci est mutualisé entre tous les médecins utilisant le service de RV.
- le secrétariat et tous les médecins sont reliés à Internet
- les RV sont enregistrés dans une base de données centralisée, accessible par Internet, par le secrétariat et les médecins

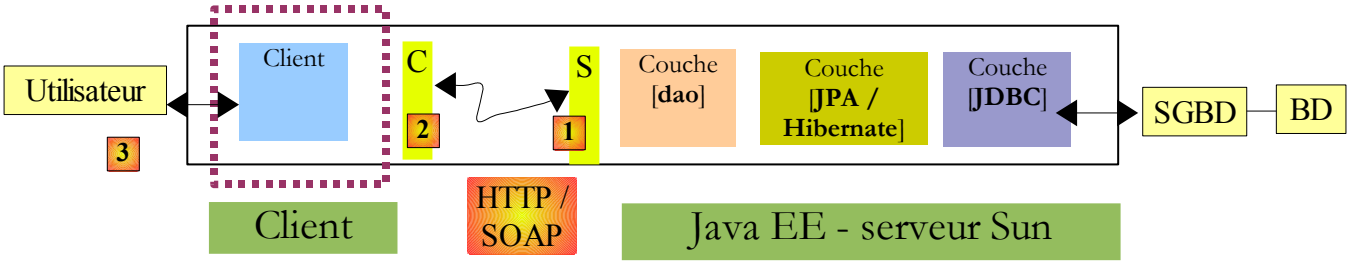
- la prise de RV est normalement faite par le secrétariat. Elle peut être faite également par les médecins eux-mêmes. C'est le cas notamment lorsqu'à la fin d'une consultation, le médecin fixe lui-même un nouveau RV à son patient.

L'architecture du service de prise de RV est le suivant :



Les médecins gagnent en efficacité s'ils n'ont plus à gérer les RV. S'ils sont suffisamment nombreux, leur contribution aux frais de fonctionnement du secrétariat sera faible.

La société [ISTIA-IAIE] décide de réaliser la partie serveur sous la forme d'un service web. L'application aura l'architecture suivante :

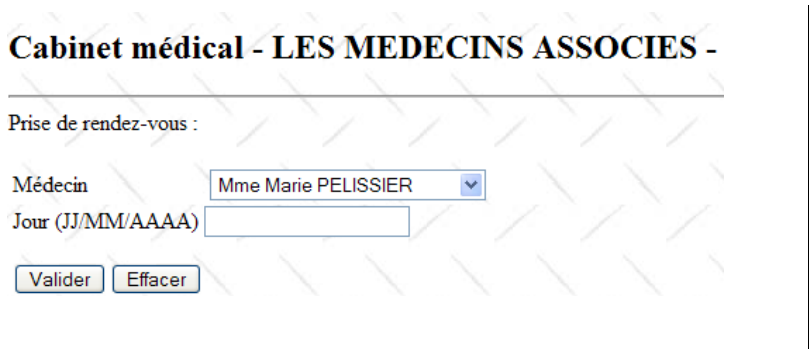


- [1] : les couches [dao, jpa] permettant l'accès aux données sont réalisées à l'aide d'un service web J2EE
- [2] : nous présenterons divers types de clients : Java, C#, Asp.Net

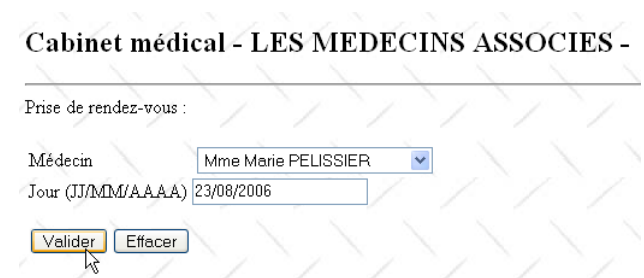
3 Fonctionnement de l'application

Nous appellerons [RdvMedecins] l'application. Nous présentons ci-dessous des copies d'écran de ce que pourrait être son fonctionnement avec un client web. Ce client web n'est pas présenté dans ce tutoriel. On le trouvera dans le document [http://tahe.developpez.com/dotnet/exercices/], exercice n° 4.

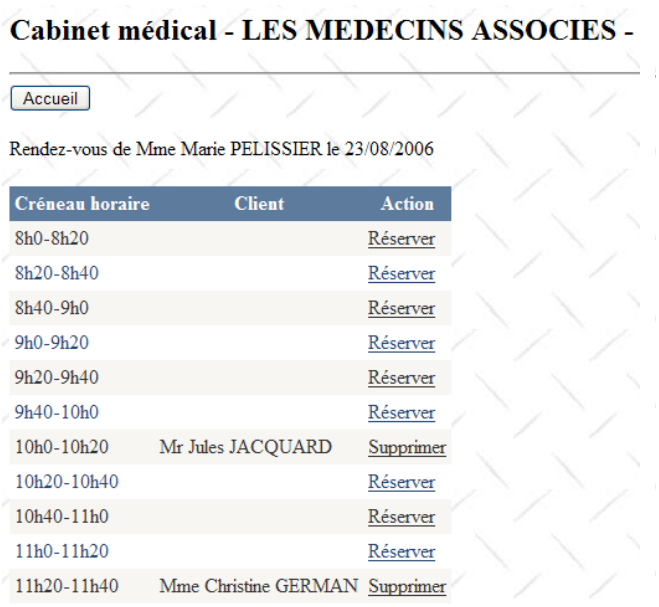
La page d'accueil de l'application est la suivante :



A partir de cette première page, l'utilisateur (Secrétariat, Médecin) va engager un certain nombre d'actions. Nous les présentons ci-dessous. La vue de gauche présente la vue à partir de laquelle l'utilisateur fait une **demande**, la vue de droite la **réponse** envoyée par le serveur.

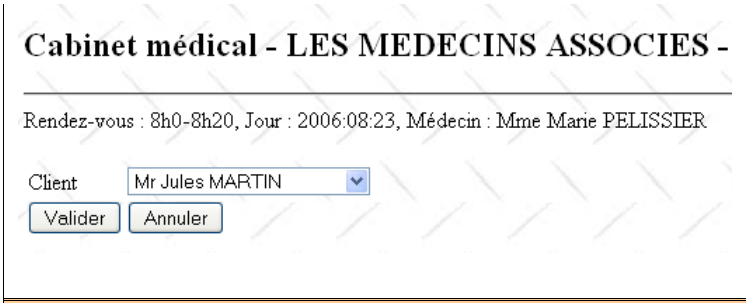


L'utilisateur a sélectionné un médecin et a saisi un jour de RV



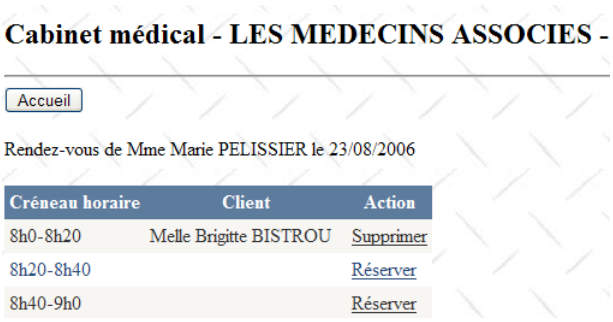
On obtient la liste (vue partielle ici) des RV du médecin sélectionné pour le jour indiqué.

On réserve

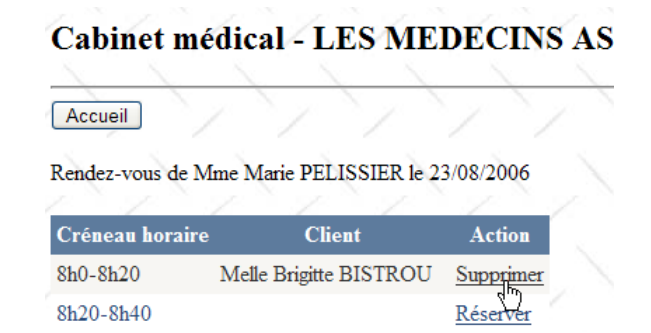


On obtient une page à renseigner

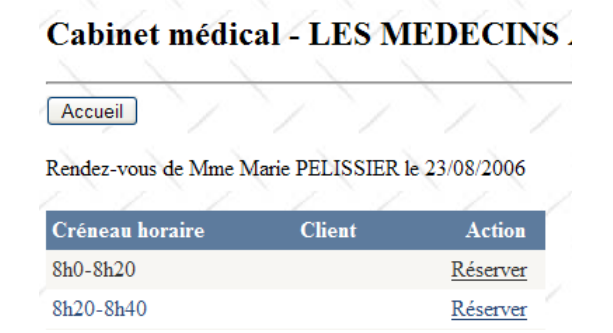
On la renseigne



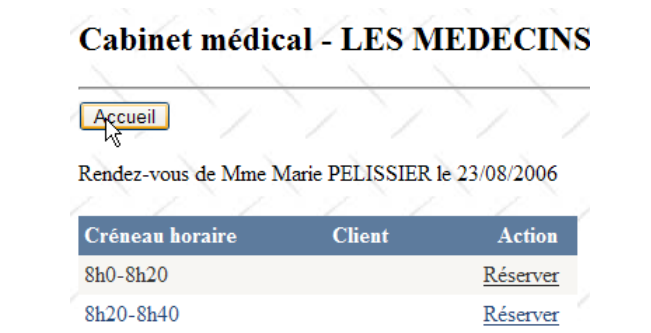
Le nouveau RV apparaît dans la liste



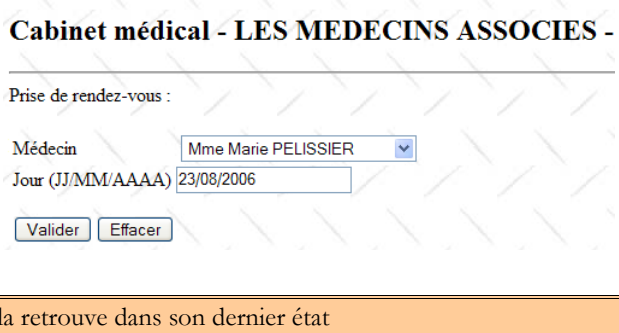
L'utilisateur peut supprimer un RV



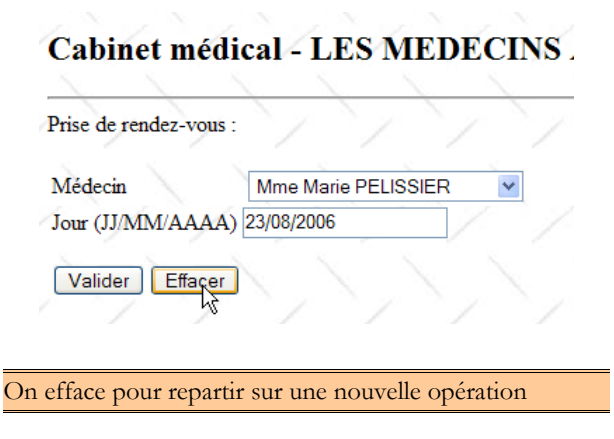
Le RV supprimé a disparu de la liste des RV

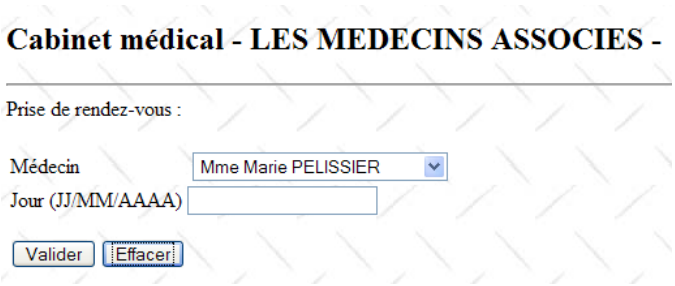


Une fois l'opération de prise de RV ou d'annulation de RV faite, l'utilisateur peut revenir à la page d'accueil



Il la retrouve dans son dernier état

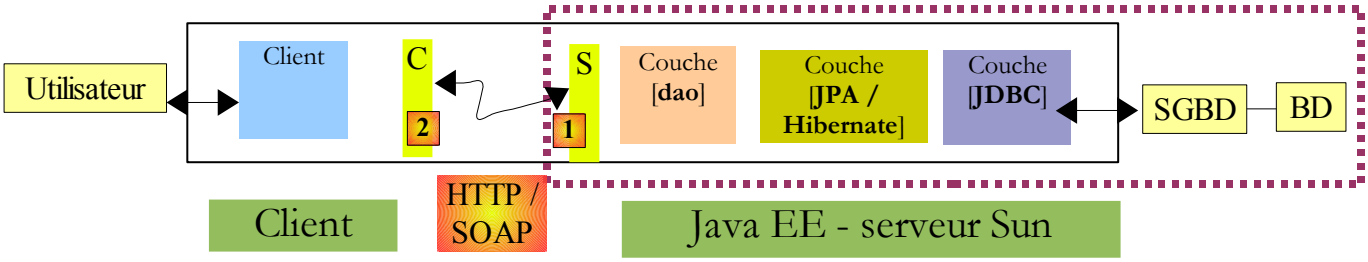




On efface pour repartir sur une nouvelle opération

4 Le service web Java EE des rendez-vous

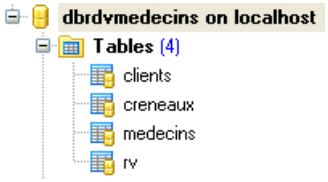
Revenons à l'architecture de l'application à construire :



Nous nous intéressons dans cette partie à la construction du service web J2EE [1] exécuté sur un serveur Sun / Glassfish.

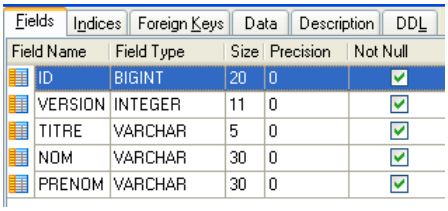
4.1 La base de données

La base de données qu'on appellera [dbrdvmedecins] est une base de données MySQL5 avec quatre tables :

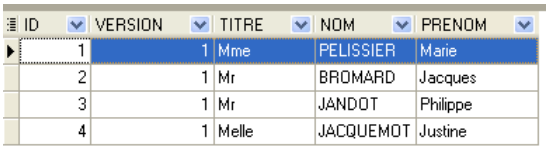


4.1.1 La table [MEDECINS]

Elle contient des informations sur les médecins gérés par l'application [RdvMedecins].



Field Name	Field Type	Size	Precision	Not Null
ID	BIGINT	20	0	☑
VERSION	INTEGER	11	0	☑
TITRE	VARCHAR	5	0	☑
NOM	VARCHAR	30	0	☑
PRENOM	VARCHAR	30	0	☑

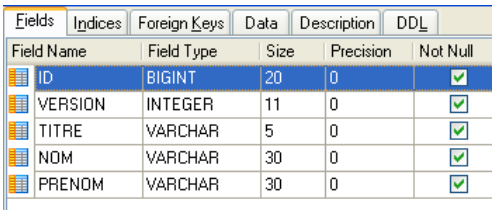


ID	VERSION	TITRE	NOM	PRENOM
1	1	Mme	PELISSIER	Marie
2	1	Mr	BROMARD	Jacques
3	1	Mr	JANDOT	Philippe
4	1	Melle	JACQUEMOT	Justine

- ID : n° identifiant le médecin - clé primaire de la table
- VERSION : n° identifiant la version de la ligne dans la table. Ce nombre est incrémenté de 1 à chaque fois qu'une modification est apportée à la ligne.
- NOM : le nom du médecin
- PRENOM : son prénom
- TITRE : son titre (Melle, Mme, Mr)

4.1.2 La table [CLIENTS]

Les clients des différents médecins sont enregistrés dans la table [CLIENTS] :



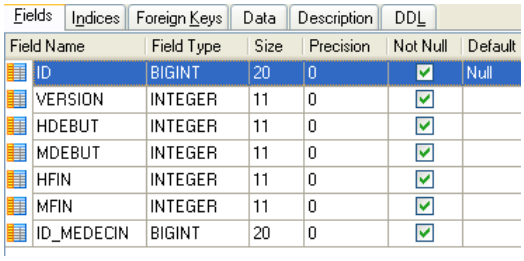
Field Name	Field Type	Size	Precision	Not Null
ID	BIGINT	20	0	☑
VERSION	INTEGER	11	0	☑
TITRE	VARCHAR	5	0	☑
NOM	VARCHAR	30	0	☑
PRENOM	VARCHAR	30	0	☑

ID	VERSION	TITRE	NOM	PRENOM
1	1	Mr	MARTIN	Jules
2	1	Mme	GERMAN	Christine
3	1	Mr	JACQUARD	Jules
4	1	Melle	BISTROU	Brigitte

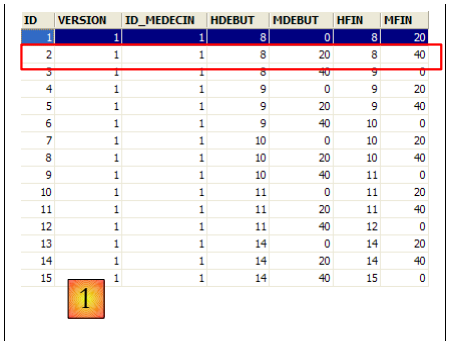
- ID : n° identifiant le client - clé primaire de la table
- VERSION : n° identifiant la version de la ligne dans la table. Ce nombre est incrémenté de 1 à chaque fois qu'une modification est apportée à la ligne.
- NOM : le nom du client
- PRENOM : son prénom
- TITRE : son titre (Melle, Mme, Mr)

4.1.3 La table [CRENEAUX]

Elle liste les créneaux horaires où les RV sont possibles :

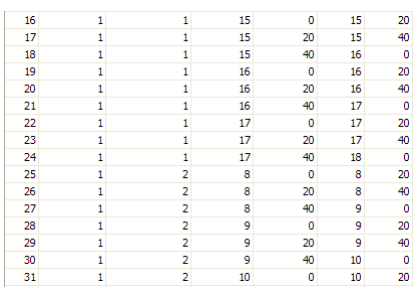


Field Name	Field Type	Size	Precision	Not Null	Default
ID	BIGINT	20	0	☑	Null
VERSION	INTEGER	11	0	☑	
HDEBUT	INTEGER	11	0	☑	
MDEBUT	INTEGER	11	0	☑	
HFIN	INTEGER	11	0	☑	
MFIN	INTEGER	11	0	☑	
ID_MEDECIN	BIGINT	20	0	☑	

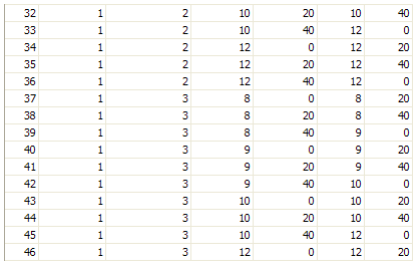


ID	VERSION	ID_MEDECIN	HDEBUT	MDEBUT	HFIN	MFIN
1	1	1	8	0	8	20
2	1	1	8	20	8	40
3	1	1	8	40	9	0
4	1	1	9	0	9	20
5	1	1	9	20	9	40
6	1	1	9	40	10	0
7	1	1	10	0	10	20
8	1	1	10	20	10	40
9	1	1	10	40	11	0
10	1	1	11	0	11	20
11	1	1	11	20	11	40
12	1	1	11	40	12	0
13	1	1	14	0	14	20
14	1	1	14	20	14	40
15	1	1	14	40	15	0

1



16	1	1	15	0	15	20
17	1	1	15	20	15	40
18	1	1	15	40	16	0
19	1	1	16	0	16	20
20	1	1	16	20	16	40
21	1	1	16	40	17	0
22	1	1	17	0	17	20
23	1	1	17	20	17	40
24	1	1	17	40	18	0
25	1	2	8	0	8	20
26	1	2	8	20	8	40
27	1	2	8	40	9	0
28	1	2	9	0	9	20
29	1	2	9	20	9	40
30	1	2	9	40	10	0
31	1	2	10	0	10	20



32	1	2	10	20	10	40
33	1	2	10	40	12	0
34	1	2	12	0	12	20
35	1	2	12	20	12	40
36	1	2	12	40	12	0
37	1	3	8	0	8	20
38	1	3	8	20	8	40
39	1	3	8	40	9	0
40	1	3	9	0	9	20
41	1	3	9	20	9	40
42	1	3	9	40	10	0
43	1	3	10	0	10	20
44	1	3	10	20	10	40
45	1	3	10	40	12	0
46	1	3	12	0	12	20

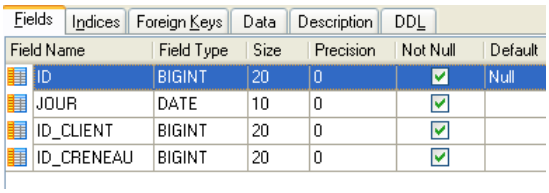
- ID : n° identifiant le créneau horaire - clé primaire de la table (ligne 8)
- VERSION : n° identifiant la version de la ligne dans la table. Ce nombre est incrémenté de 1 à chaque fois qu'une modification est apportée à la ligne.
- ID_MEDECIN : n° identifiant le médecin auquel appartient ce créneau – clé étrangère sur la colonne MEDECINS(ID).
- HDEBUT : heure début créneau
- MDEBUT : minutes début créneau

- HFIN : heure fin créneau
- MFIN : minutes fin créneau

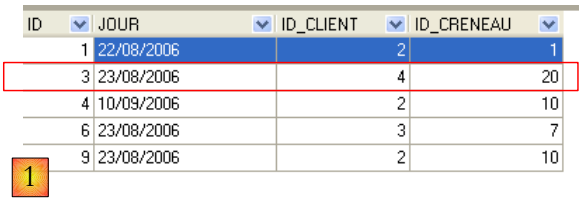
La seconde ligne de la table [CRENEAUX] (cf [1] ci-dessus) indique, par exemple, que le créneau n° 2 commence à 8 h 20 et se termine à 8 h 40 et appartient au médecin n° 1 (Mme Marie PELISSIER).

4.1.4 La table [RV]

Elle liste les RV pris pour chaque médecin :



Field Name	Field Type	Size	Precision	Not Null	Default
ID	BIGINT	20	0	☑	Null
JOUR	DATE	10	0	☑	
ID_CLIENT	BIGINT	20	0	☑	
ID_CRENEAU	BIGINT	20	0	☑	



ID	JOUR	ID_CLIENT	ID_CRENEAU
1	22/08/2006	2	1
3	23/08/2006	4	20
4	10/09/2006	2	10
6	23/08/2006	3	7
9	23/08/2006	2	10

- ID : n° identifiant le RV de façon unique – clé primaire
- JOUR : jour du RV
- ID_CRENEAU : créneau horaire du RV - clé étrangère sur le champ [ID] de la table [CRENEAUX] – fixe à la fois le créneau horaire et le médecin concerné.
- ID_CLIENT : n° du client pour qui est faite la réservation – clé étrangère sur le champ [ID] de la table [CLIENTS]

Cette table a une contrainte d'unicité sur les valeurs des colonnes jointes (JOUR, ID_CRENEAU) :

```
ALTER TABLE RV ADD CONSTRAINT UNQ1_RV UNIQUE (JOUR, ID_CRENEAU);
```

Si une ligne de la table[RV] a la valeur (JOUR1, ID_CRENEAU1) pour les colonnes (JOUR, ID_CRENEAU), cette valeur ne peut se retrouver nulle part ailleurs. Sinon, cela signifierait que deux RV ont été pris au même moment pour le même médecin. D'un point de vue programmation Java, le pilote JDBC de la base lance une *SQLException* lorsque ce cas se produit.

La ligne d'*id* égal à 3 (cf [1] ci-dessus) signifie qu'un RV a été pris pour le créneau n° 20 et le client n° 4 le 23/08/2006. La table [CRENEAUX] nous apprend que le créneau n° 20 correspond au créneau horaire 16 h 20 - 16 h 40 et appartient au médecin n° 1 (Mme Marie PELISSIER). La table [CLIENTS] nous apprend que le client n° 4 est Melle Brigitte BISTROU.

4.2 Génération de la base de données

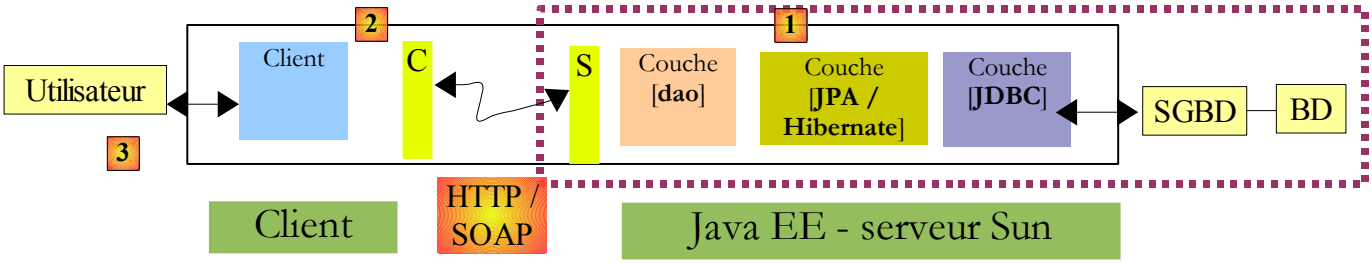
Créez la base de données MySql [dbrdvmedecins] avec l'outil de votre choix. Pour créer les tables et les remplir on pourra utiliser le script [createbd.sql] qui vous sera fourni. Son contenu est le suivant :

```
1.  create table CLIENTS (
2.          ID bigint not null auto_increment,
3.          VERSION integer not null,
4.          TITRE varchar(5) not null,
5.          NOM varchar(30) not null,
6.          PRENOM varchar(30) not null,
7.          primary key (ID)
8.      ) ENGINE=InnoDB;
9.
10.     create table CRENEAUX (
11.         ID bigint not null auto_increment,
12.         VERSION integer not null,
13.         HDEBUT integer not null,
14.         MDEBUT integer not null,
15.         HFIN integer not null,
16.         MFIN integer not null,
17.         ID_MEDECIN bigint not null,
18.         primary key (ID)
19.     ) ENGINE=InnoDB;
20.
21.     create table MEDECINS (
22.         ID bigint not null auto_increment,
```

```
23.         VERSION integer not null,
24.         TITRE varchar(5) not null,
25.         NOM varchar(30) not null,
26.         PRENOM varchar(30) not null,
27.         primary key (ID)
28.     ) ENGINE=InnoDB;
29.
30.     create table RV (
31.         ID bigint not null auto_increment,
32.         JOUR date not null,
33.         ID_CLIENT bigint not null,
34.         ID_CRENEAU bigint not null,
35.         primary key (ID)
36.     ) ENGINE=InnoDB;
37.
38.     alter table CRENEAUX
39.         add index FK9BD7A197FE16862 (ID_MEDECIN),
40.         add constraint FK9BD7A197FE16862
41.         foreign key (ID_MEDECIN)
42.         references MEDECINS (ID);
43.
44.     alter table RV
45.         add index FKA4494D97AD2 (ID_CLIENT),
46.         add constraint FKA4494D97AD2
47.         foreign key (ID_CLIENT)
48.         references CLIENTS (ID);
49.
50.     alter table RV
51.         add index FKA441A673246 (ID_CRENEAU),
52.         add constraint FKA441A673246
53.         foreign key (ID_CRENEAU)
54.         references CRENEAUX (ID);
55.
56. INSERT INTO CLIENTS ( VERSION, NOM, PRENOM, TITRE) VALUES (1, 'MARTIN', 'Jules', 'Mr');
57. ...
58.
59. INSERT INTO MEDECINS ( VERSION, NOM, PRENOM, TITRE) VALUES (1, 'PELISSIER', 'Marie', 'Mme');
60. ...
61.
62. INSERT INTO CRENEAUX ( VERSION, ID_MEDECIN, HDEBUT, MDEBUT, HFIN, MFIN) VALUES (1, 1, 8, 0, 8,
    20);
63. ...
64.
65. INSERT INTO RV ( JOUR, ID_CRENEAU, ID_CLIENT) VALUES ('2006-08-22', 1, 2);
66. ...
67.
68. ALTER TABLE RV ADD CONSTRAINT UNQ1_RV UNIQUE (JOUR, ID_CRENEAU);
69.
70. COMMIT WORK;
```

4.3 Les éléments de l'architecture côté serveur

Revenons à l'architecture de l'application à construire :



Côté serveur, l'application sera formée :

(a) d'une couche Jpa permettant de travailler avec la BD au moyen d'objets
(b) d'un Ejb chargé de gérer les opérations avec la couche Jpa
(c) d'un service web chargé d'exposer à des clients distants, l'interface de l'Ejb sous la forme d'un service web.

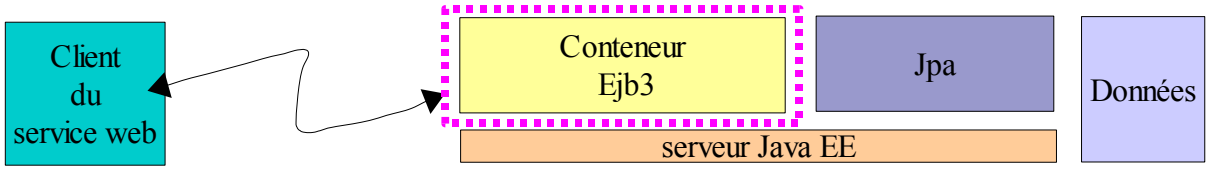
Les éléments (b) et (c) impémentent la couche [dao] représentée sur le schéma précédent. On sait qu'une application peut accéder à un Ejb distant via les protocoles RMI et JNDI. Dans la pratique, cela limite les clients à des clients Java. Un service web utilise un protocole de communication standardisé que divers langages implémentent : .NET, Php, C++, ... C'est ce que nous voulons montrer ici en utilisant un client .NET.

Pour une courte introduction aux services web, on pourra lire le cours [**ref1**], paragraphe 14, page 109.

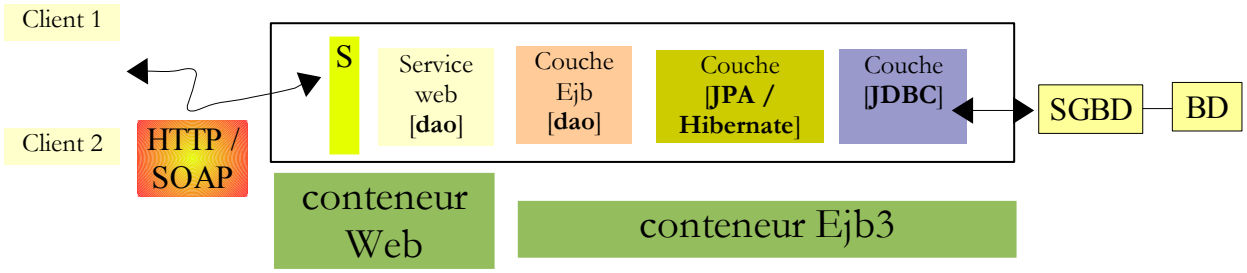
Un service web peut être implémenté de deux façons :

- par une classe annotée **@WebService** qui s'exécute dans un conteneur web

- par un Ejb annoté **@WebService** qui s'exécute dans un conteneur Ejb



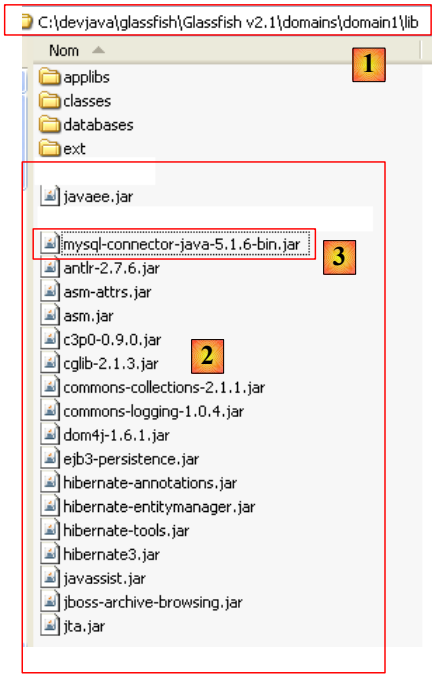
Nous allons utiliser ici la première solution :



Dans le cours [**ref1**], paragraphe 14, page 109, on trouvera un exemple utilisant la seconde solution.

4.4 Configuration Hibernate du serveur Glassfish

Selon sa version, le serveur Glassfish V2 livré avec Netbeans peut ne pas avoir les bibliothèques Hibernate dont la couche Jpa / Hibernate a besoin. Si dans la suite du tutoriel, vous découvrez que Glassfish ne vous propose pas d'implémentation Jpa / Hibernate ou qu'au déploiement des services, une exception indique que les bibliothèques d'Hibernate ne sont pas trouvées, vous devez rajouter les bibliothèques dans le dossier [<glassfish>/domains/domain1/lib] puis redémarrer le serveur Glassfish :

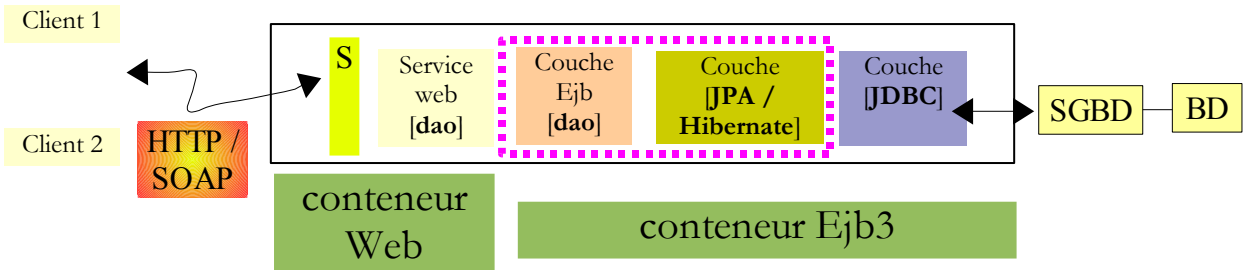


- en [1], le dossier <glassfish>/.../lib
- en [2], les bibliothèques Hibernate
- en [3], le pilote Jdbc de MySQL

Les bibliothèques d'Hibernate sont dans le zip qui accompagne le tutoriel.

4.5 Les outils de génération automatique de Netbeans

Revenons à l'architecture que nous devons construire :

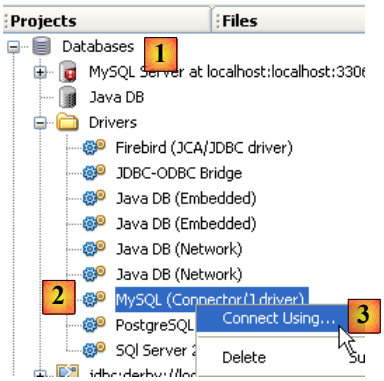


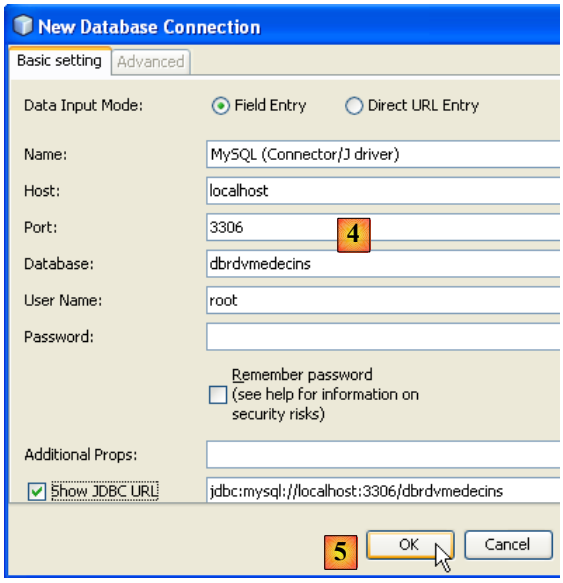
Avec Netbeans, il est possible de générer automatiquement la couche [JPA] et la couche [Ejb] qui contrôle l'accès aux entités JPA générées. Il est intéressant de connaître ces méthodes de génération automatique car le code généré donne de précieuses indications sur la façon d'écrire des entités JPA ou le code Ejb qui les utilise.

Nous décrivons maintenant certains de ces outils de génération automatique. Pour comprendre le code généré, il faut avoir de bonnes notions sur les entités JPA [**ref1**] et les EJB [**ref2**].

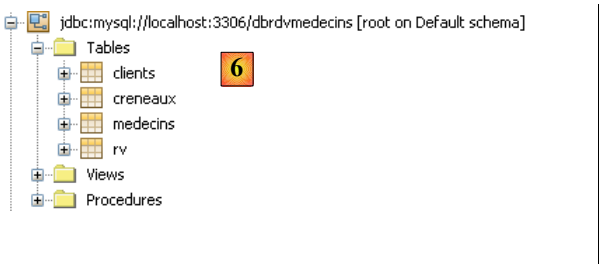
Création d'une connexion Netbeans à la base de données

- lancer le SGBD MySQL 5 afin que la BD soit disponible
- créer une connexion Netbeans sur la base [dbrdvmedecins]



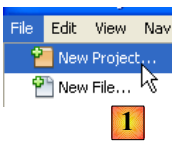


- dans l'onglet [Files], dans la branche [Databases] [1], sélectionner le pilote Jdbc MySQL [2]
- puis sélectionner l'option [3] "Connect Using" permettant de créer une connexion avec une base MySQL
- en [4], donner les informations qui vous sont demandées
- puis valider en [5]

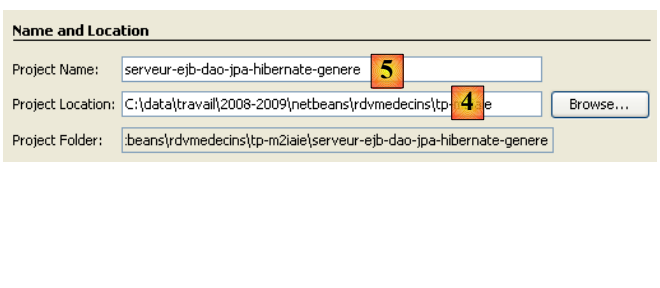


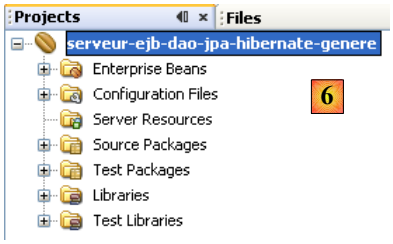
- en [6], la connexion est créée. On y voit les quatre tables de la base de données connectée.

Création d'un projet Ejb



- en [1], créer une nouvelle application, un module Ejb
- en [2], choisir la catégorie [Java EE] et en [3] le type [EJB Module]

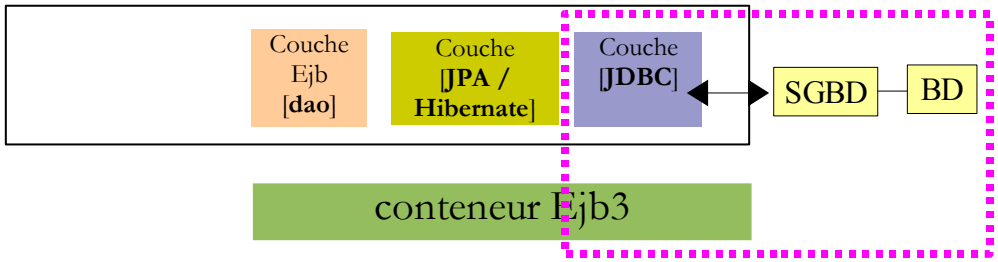


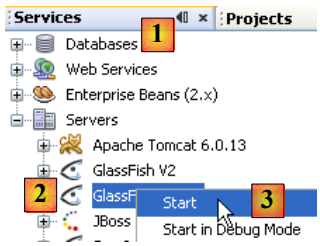


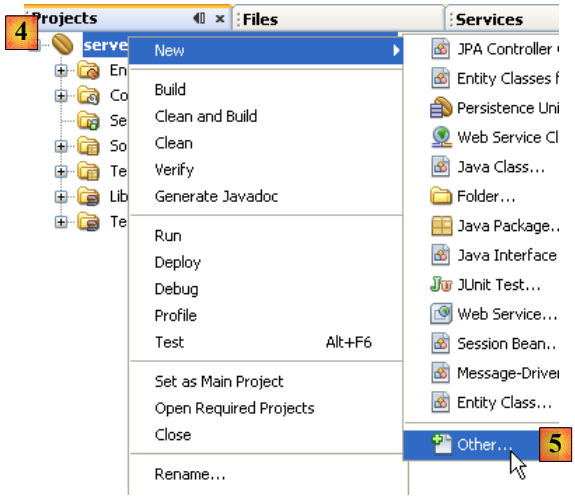
- en [4] choisir un dossier pour le projet et en [5] lui donner un nom - puis terminer l'assistant
- en [6] le projet généré

Ajout d'une ressource JDBC au serveur Glassfish

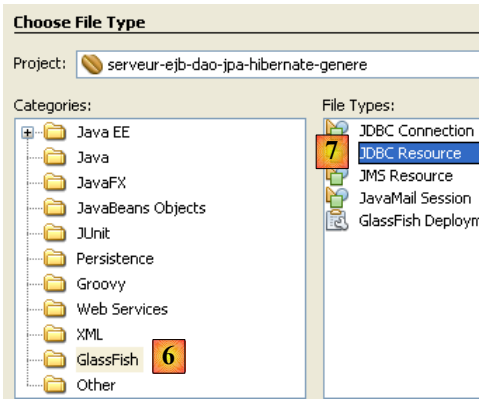
Nous allons ajouter une ressource JDBC au serveur Glassfish.

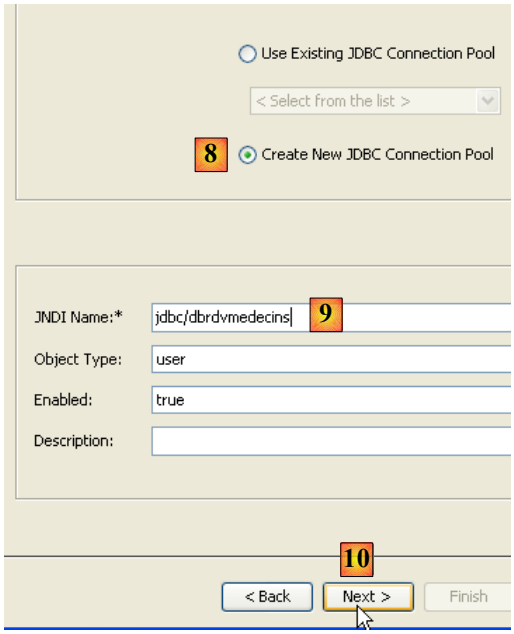




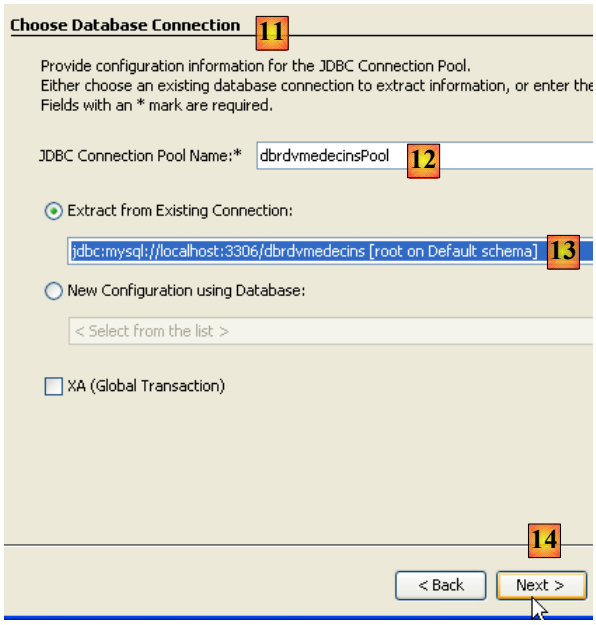


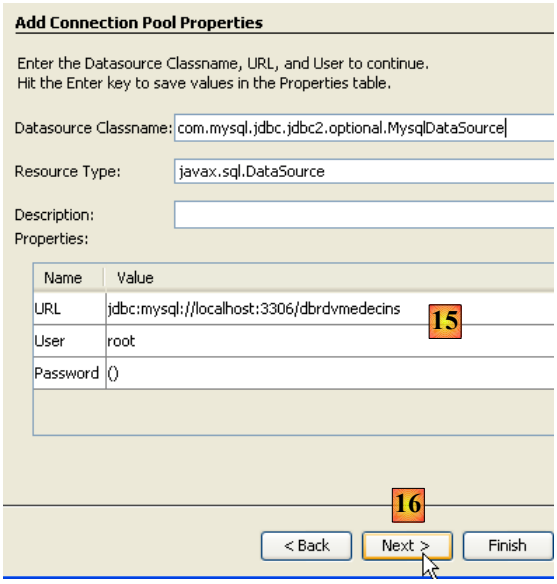
- dans l'onglet [Services], lancer le serveur Glassfish [2, 3]
- dans l'onglet [Projects], cliquer droit sur le projet Ejb et en [5] sélectionner l'option [New / Other] permettant d'ajouter un élément au projet.





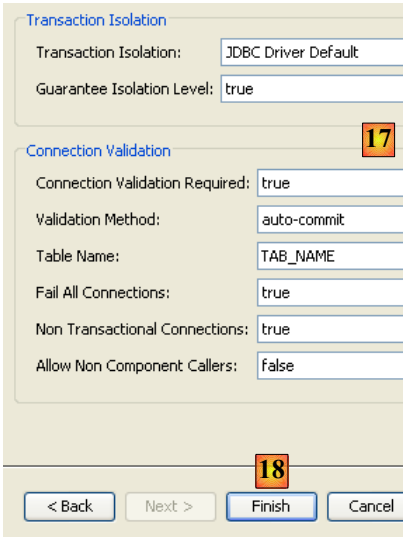
- en [6], sélectionner la catégorie [Glassfish] et en [7] indiquer qu'on veut créer une ressource JDBC en sélectionnant le type [JDBC Resource]
- en [8], indiquer que cette ressource JDBC va utiliser son propre pool de connexions
- en [9], donner un nom à la ressource JDBC
- en [10], passer à l'étape suivante

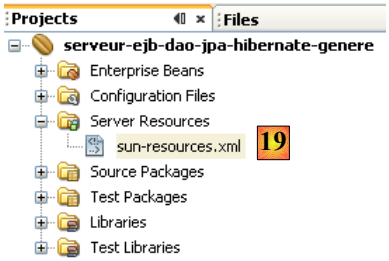




- en [11], on définit les caractéristiques du pool de connexions de la ressource JDBC
- en [12], donner un nom au pool de connexions
- en [13], choisir la connexion Netbeans [dbrdvmedecins] créée précédemment
- en [14], passer à l'étape suivante
- en [15], il n'y a normalement rien à changer dans cette page. Les propriétés de la connexion à la base MySQL [dbrdvmedecins] ont été tirées de celles de la connexion Netbeans [dbrdvmedecins] créée précédemment

- en [16], passer à l'étape suivante





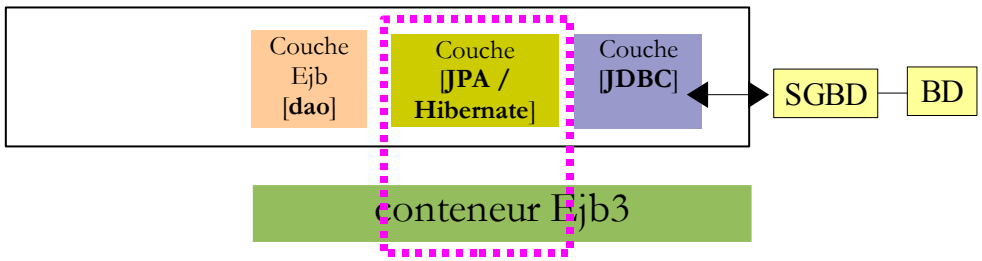
- en [17], garder les valeurs par défaut proposées
- en [18], terminer l'assistant. Celui crée le fichier [sun-resources.xml] [19] dont le contenu est le suivant :

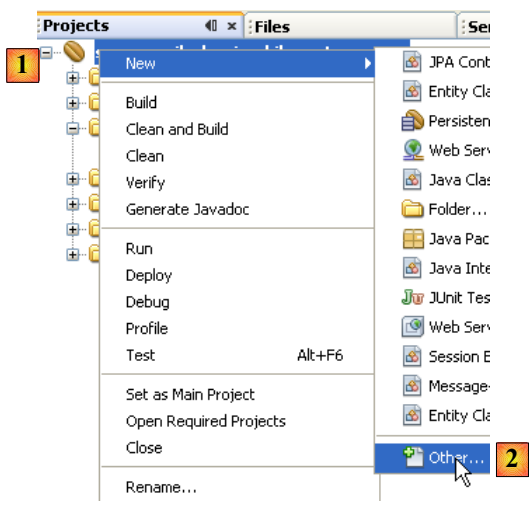
```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <!DOCTYPE resources PUBLIC "-//Sun Microsystems, Inc.//DTD Application Server 9.0 Resource
    Definitions //EN" "http://www.sun.com/software/appserver/dtds/sun-resources_1_3.dtd">
3.  <resources>
4.    <jdbc-resource enabled="true" jndi-name="jdbc/dbrdvmedecins" object-type="user" pool-
    name="dbrdvmedecinsPool">
5.      <description/>
6.    </jdbc-resource>
7.    <jdbc-connection-pool ...">
8.      <property name="URL" value="jdbc:mysql://localhost:3306/dbrdvmedecins"/>
9.      <property name="User" value="root"/>
10.     <property name="Password" value="()"/>
11.   </jdbc-connection-pool>
12. </resources>
```

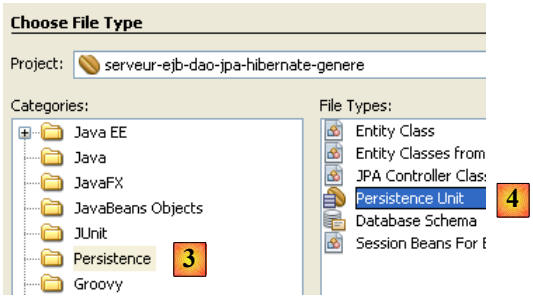
Le fichier ci-dessus reprend toutes les informations saisies dans l'assistant sous un format XML. Il sera utilisé par l'IDE Netbeans pour demander au serveur Glassfish de créer la ressource "jdbc/dbrdvmedecins" définie ligne 4.

Création d'une unité de persistance

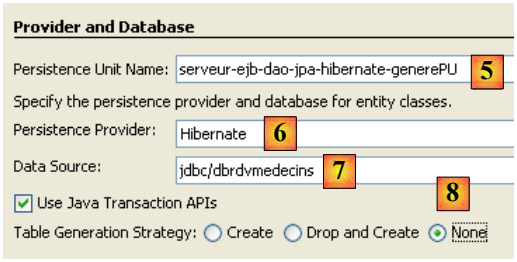
L'unité de persistance [persistence.xml] configure la couche JPA : elle indique l'implémentation JPA utilisée (Toplink, Hibernate, ...) et configure celle-ci.

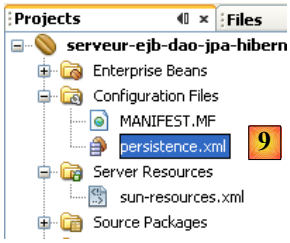






- en [1], cliquer droit sur le projet Ejb et sélectionner [New / Other] en [2]
- en [3], sélectionner la catégorie [Persistence] puis en [4], indiquer que vous voulez créer une unité de persistance JPA





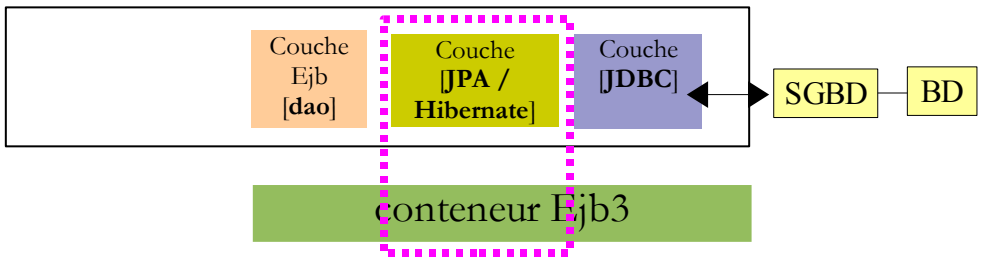
- en [5], donner un nom à l'unité de persistance créée
- en [6], choisir [Hibernate] comme implémentation JPA
- en [7], sélectionner la ressource Glassfish "jdbc/dbrdvmedecins" qui vient d'être créée
- en [8], indiquer qu'aucune action ne doit être faite sur la base, lors de l'instanciation de la couche JPA
- terminer l'assistant
- en [9], le fichier [persistence.xml] créé par l'assistant

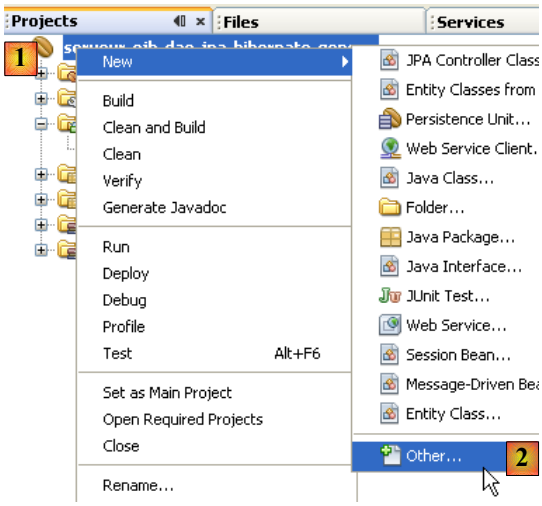
Son contenu est le suivant :

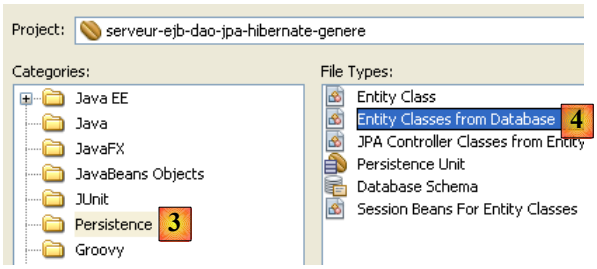
```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/
    ns/persistence http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
3.    <persistence-unit name="serveur-ejb-dao-jpa-hibernate-generePU" transaction-type="JTA">
4.      <provider>org.hibernate.ejb.HibernatePersistence</provider>
5.      <jta-data-source>jdbc/dbrdvmedecins</jta-data-source>
6.      <exclude-unlisted-classes>false</exclude-unlisted-classes>
7.      <properties/>
8.    </persistence-unit>
9.  </persistence>
```

De nouveau, il reprend dans un format XML les informations données dans l'assistant. Ce fichier est insuffisant pour travailler avec la base MySQL5 "dbrdvmedecins". Il nous faudrait indiquer à Hibernate le type de SGBD à gérer. Ce sera fait ultérieurement.

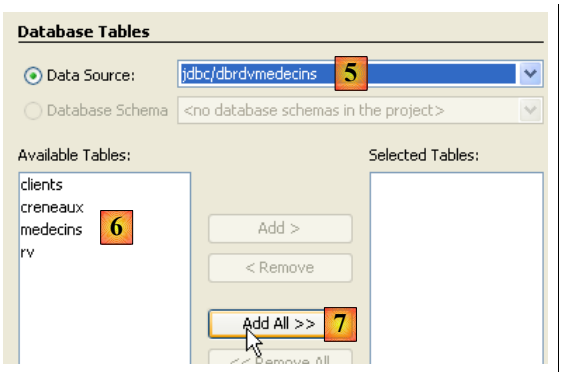
Création des entités JPA

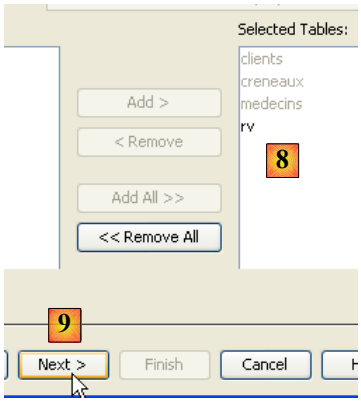




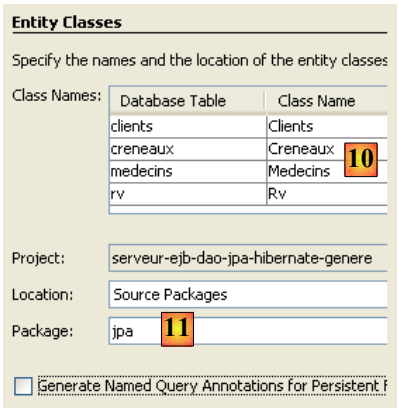


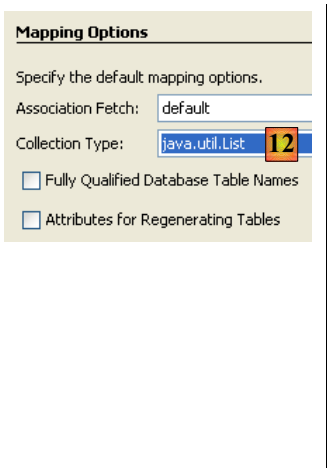
- en [1], cliquer droit sur le projet et en [2] choisir l'option [New / Other]
- en [3], sélectionner la catégorie [Persistence] puis en [4], indiquer que vous voulez créer des entités JPA à partir d'une base de données existante.

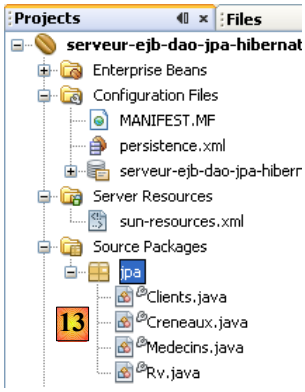




- en [5], sélectionner la source JDBC "jdbc/dbrdvmedecins" que nous avons créée
- en [6], les quatre tables de la base de données associée
- en [7,8], les inclure toutes dans la génération des entités JPA
- en [9], poursuivre l'assistant





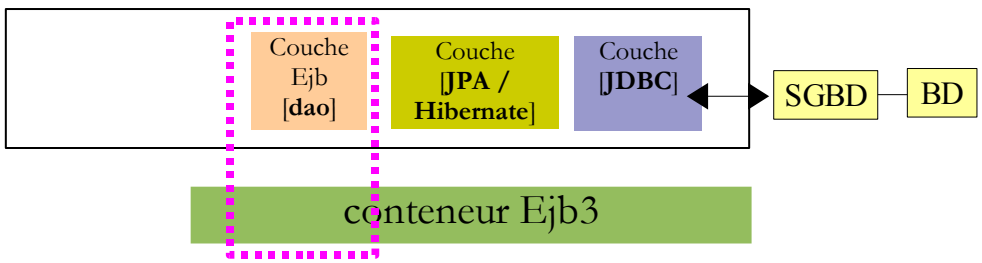


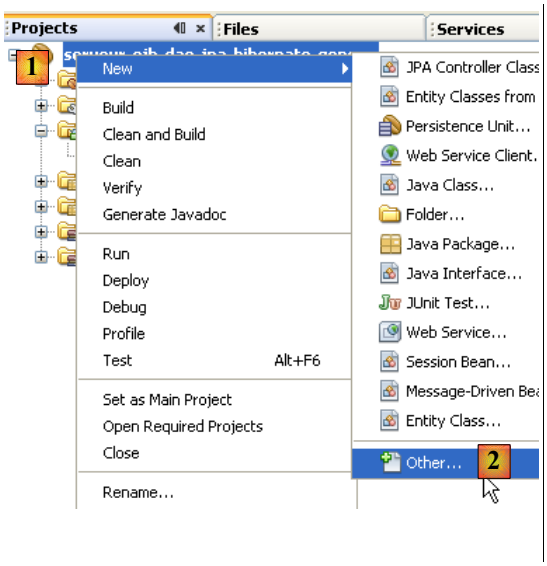
- en [10], les entités JPA qui vont être générées
- en [11], donner un nom au package des entités JPA
- en [12], choisir le type Java qui va encapsuler les listes d'objets rendus par la couche JPA
- terminer l'assistant
- en [13], les quatre entités JPA générées, une pour chaque table de la base de données.

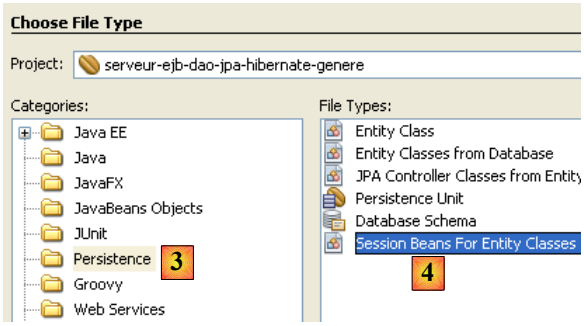
Voici par exemple le code de l'entité [Rv] qui représente une ligne de la table [rv] de la base [dbrdvmedecins].

```
1.  package jpa;
2.  ...
3.  @Entity
4.  @Table(name = "rv")
5.  public class Rv implements Serializable {
6.    private static final long serialVersionUID = 1L;
7.    @Id
8.    @GeneratedValue(strategy = GenerationType.IDENTITY)
9.    @Basic(optional = false)
10.   @Column(name = "ID")
11.   private Long id;
12.   @Basic(optional = false)
13.   @Column(name = "JOUR")
14.   @Temporal(TemporalType.DATE)
15.   private Date jour;
16.   @JoinColumn(name = "ID_CRENEAU", referencedColumnName = "ID")
17.   @ManyToOne(optional = false)
18.   private Creneaux idCreneau;
19.   @JoinColumn(name = "ID_CLIENT", referencedColumnName = "ID")
20.   @ManyToOne(optional = false)
21.   private Clients idClient;
22.
23.   public Rv() {
24.   }
25.
26. ...
27. }
```

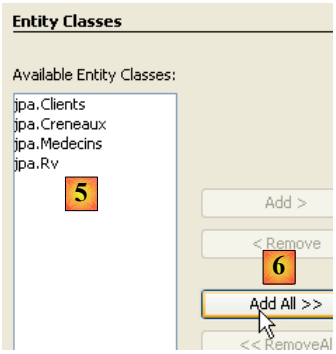
Création de la couche Ejb d'accès aux entités JPA

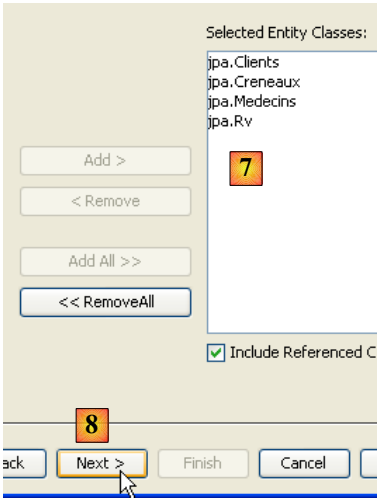




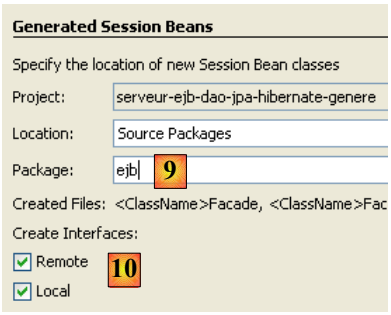


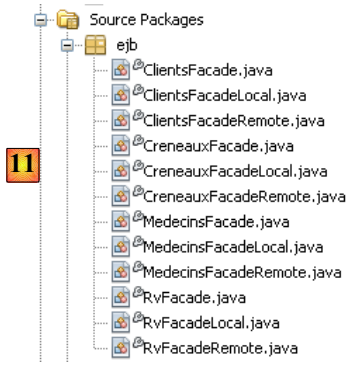
- en [1], cliquer droit sur le projet et en [2], sélectionner l'option [New / Other]
- en [3], sélectionner la catégorie [Persistence] puis en [4] le type [Session Beans for Entity Classes]





- en [5], les entités JPA créées précédemment sont présentées
- en [6], les sélectionner toutes
- en [7], elles ont été sélectionnées
- en [8], poursuivre l'assistant





- en [9], donner un nom au package des ejb qui vont être générés
- en [10], indiquer que les Ejb doivent implémenter à la fois une interface locale et distante
- terminer l'assistant
- en [11], les Ejb générés

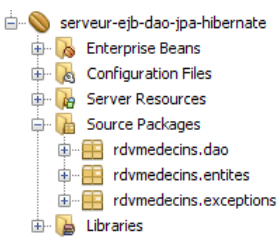
Voici par exemple, le code de l'Ejb qui gère l'accès à l'entité [Rv], donc à la table [rv] de la base de données [dbrdvmedecins] :

```
1.  package ejb;
2.  ...
3.  @Stateless
4.  public class RvFacade implements RvFacadeLocal, RvFacadeRemote {
5.    @PersistenceContext
6.    private EntityManager em;
7.
8.    public void create(Rv rv) {
9.      em.persist(rv);
10.   }
11.
12.   public void edit(Rv rv) {
13.     em.merge(rv);
14.   }
15.
16.   public void remove(Rv rv) {
17.     em.remove(em.merge(rv));
18.   }
19.
20.   public Rv find(Object id) {
21.     return em.find(Rv.class, id);
22.   }
23.
24.   public List<Rv> findAll() {
25.     return em.createQuery("select object(o) from Rv as o").getResultList();
26.   }
27.
28. }
```

Comme il a été dit, la génération automatique de code peut être très utile pour démarrer un projet et se former sur les entités JPA et les EJB. Dans la suite, nous réécrivons les couches JPA et EJB avec notre propre code mais le lecteur y retrouvera des informations que nous venons de voir dans la génération automatique des couches.

4.6 Le projet Netbeans du module Ejb

Nous créons un nouveau module Ejb vierge (cf paragraphe 4.5, page 12) :

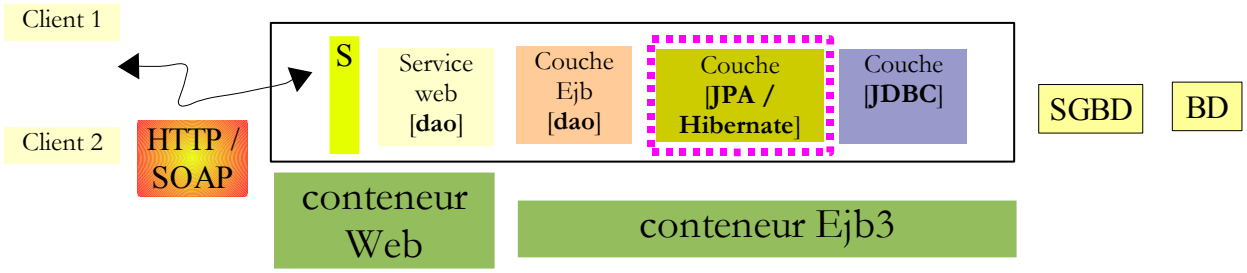


- le package [rdvmedecins.entites] regroupe les entités de la couche Jpa
- le package [rdvmedecins.dao] implémente l'Ejb de la couche [dao]
- le package [rdvmedecins.exceptions] implémente une classe d'exception spécifique à l'application

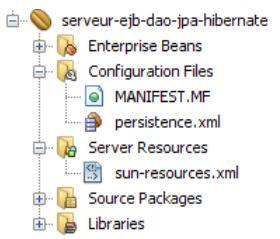
Dans la suite, nous supposons que le lecteur a suivi toutes les étapes du paragraphe 4.5, page 11. Il devra en répéter certaines.

4.6.1 Configuration de la couche JPA

Rappelons l'architecture de notre application client / serveur :



Le projet Netbeans :



La couche [JPA] est configurée par les fichiers [persistence.xml] et [sun-resources.xml] ci-dessus. Ces deux fichiers sont générés par des assistants déjà rencontrés :

- la génération du fichier [sun-resources.xml] a été décrite au pararaphe 4.5, page 13.
- la génération du fichier [persistence.xml] a été décrite au pararaphe 4.5, page 15.

Le fichier [persistence.xml] généré doit être modifié de la façon suivante :

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <persistence version="1.0" 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_1_0.xsd">
3.    <persistence-unit name="dbrdvmedecins" transaction-type="JTA">
4.      <provider>org.hibernate.ejb.HibernatePersistence</provider>
5.      <jta-data-source>jdbc/dbrdvmedecins</jta-data-source>
6.      <properties>
7.        <!-- Dialecte -->
```

```
          <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5InnoDBDialect"/>
8.      </properties>
9.    </persistence-unit>
10. </persistence>
```

- ligne 3 : le type de transactions est JTA : les transactions seront gérées par le conteneur Ejb3 de Glassfish
- ligne 4 : une implémentation Jpa / Hibernate est utilisée. Pour cela, la bibliothèque Hibernate a été ajoutée au serveur Glassfish (cf paragraphe 4.4, page 10).
- ligne 5 : la source de données JTA utilisée par la couche Jpa a le nom JNDI « jdbc/dbrdvmedecins ».
- ligne 8 : cette ligne n'est pas générée automatiquement. Elle doit être ajoutée à la main. Elle indique à Hibernate, que le SGBD utilisé est MySQL5.

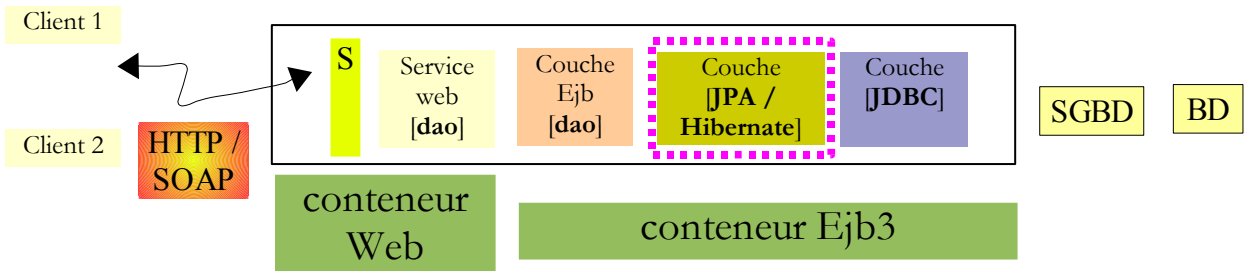
La source de données "**jdbc/dbrdvmedecins"** est configurée dans le fichier [sun-resources.xml] suivant :

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <!DOCTYPE resources PUBLIC "-//Sun Microsystems, Inc.//DTD Application Server 9.0 Resource
    Definitions //EN" "http://www.sun.com/software/appserver/dtds/sun-resources_1_3.dtd">
3.  <resources>
4.    <jdbc-resource enabled="true" jndi-name="jdbc/dbrdvmedecins" object-type="user" pool-
    name="dbrdvmedecinsPool">
5.      <description/>
6.    </jdbc-resource>
7.    <jdbc-connection-pool ...>
8.      <property name="URL" value="jdbc:mysql://localhost/dbrdvmedecins"/>
9.      <property name="User" value="root"/>
10.     <property name="Password" value="()"/>
11.   </jdbc-connection-pool>
12. </resources>
```

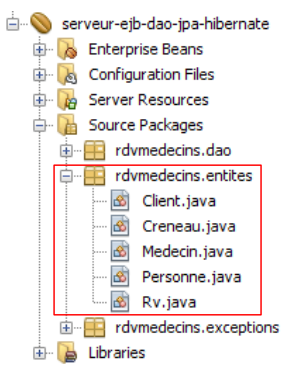
- lignes 8-10 : les caractéristiques Jdbc de la source de données (Url de la base, nom et mot de passe de l'utilisateur). La base de données MySQL *dbrdvmedecins* est celle décrite au paragraphe 4.1, page 6.
- ligne 7 : les caractéristiques du pool de connexions associé à cette source de données

4.6.2 Les entités de la couche JPA

Rappelons l'architecture de notre application client / serveur :



Le projet Netbeans :



Le package [rdvmedecins.entites] implémente la couche [Jpa].

Nous avons vu au paragraphe 4.5, page 16 comment générer automatiquement les entités Jpa d'une application. Nous n'utiliserons pas ici cette technique mais définirons nous-mêmes les entités. Celles-ci reprendront cependant une bonne partie du code généré au paragraphe 4.5, page 16. Ici, nous souhaitons que les entités [Medecin] et [Client] soient des classes filles d'une classe [Personne].

La classe **Personne** est utilisée pour représenter les médecins et les clients :

```
1.  package rdvmedecins.entites;
2.  ...
3.  @MappedSuperclass
4.  public class Personne implements Serializable {
5.    // caractéristiques d'une personne
6.
7.    @Id
8.    @GeneratedValue(strategy = GenerationType.AUTO)
9.    @Column(name = "ID")
10.   private Long id;
11.   @Version
12.   @Column(name = "VERSION", nullable = false)
13.   private Integer version;
14.
15.   @Column(name = "TITRE", length = 5, nullable = false)
16.   private String titre;
17.   @Column(name = "NOM", length = 30, nullable = false)
18.   private String nom;
19.   @Column(name = "PRENOM", length = 30, nullable = false)
20.   private String prenom;
21.
22.   // constructeur par défaut
23.   public Personne() {
24.   }
25.
26.   // constructeur avec paramètres
27.   public Personne(String titre, String nom, String prenom) {
28.     // on passe par les setters
29. ...
30.   }
31.
32.   // constructeur par recopie
33.   public Personne(Personne personne) {
34.     // on passe par les setters
35.  ...
36.   }
37.
38.   // toString
39.   @Override
40.   public String toString() {
41.     return "[" + titre + "," + prenom + "," + nom + "]";
42.   }
43.
44. // getters et setters
45. ....
46. }
```

- ligne 3 : on notera que la classe [Personne] n'est pas elle-même une entité (@Entity). Elle va être la classe parent d'entités. L'annotation **@MappedSuperClass** désigne cette situation.

L'entité [**Client**] encapsule les lignes de la table [clients]. Elle dérive de la classe [Personne] précédente :

```
1.  package rdvmedecins.entites;
2.  ....
3.  @Entity
4.  @Table(name = "CLIENTS")
5.  public class Client extends Personne implements Serializable {
6.
7.    // constructeur par défaut
8.    public Client() {
9.    }
10.
11.   // constructeur avec paramètres
12.   public Client(String titre, String nom, String prenom) {
13.     // parent
14.     super(titre, nom, prenom);
15.   }
```

```
16.
17.   // constructeur par recopie
18.   public Client(Client client) {
19.     // parent
20.     super(client);
21.   }
22. }
```

- ligne 3 : la classe [Client] est une entité Jpa
- ligne 4 : elle est associée à la table [clients]
- ligne 5 : elle dérive de la classe [Personne]

L'entité [**Medecin**] qui encapsule les lignes de la table [medecins] suit le même modèle :

```
1.  package rdvmedecins.entites;
2.  ...
3.  @Entity
4.  @Table(name = "MEDECINS")
5.  public class Medecin extends Personne implements Serializable {
6.
7.    // constructeur par défaut
8.    public Medecin() {
9.    }
10.
11.   // constructeur avec paramètres
12.   public Medecin(String titre, String nom, String prenom) {
13.     // parent
14.     super(titre, nom, prenom);
15.   }
16.
17.   // constructeur par recopie
18.   public Medecin(Medecin medecin) {
19.     // parent
20.     super(medecin);
21.   }
22. }
```

L'entité [**Creneau**] encapsule les lignes de la table [creneaux] :

```
1.  package rdvmedecins.entites;
2.  ....
3.  @Entity
4.  @Table(name = "CRENEAUX")
5.  public class Creneau implements Serializable {
6.
7.    // caractéristiques d'un créneau de RV
8.    @Id
9.    @GeneratedValue(strategy = GenerationType.AUTO)
10.   @Column(name = "ID")
11.   private Long id;
12.   @Version
13.   @Column(name = "VERSION", nullable = false)
14.   private Integer version;
15.   @ManyToOne
16.   @JoinColumn(name = "ID_MEDECIN", nullable = false)
17.   private Medecin medecin;
18.   @Column(name = "HDEBUT", nullable = false)
19.   private Integer hdebut;
20.   @Column(name = "MDEBUT", nullable = false)
21.   private Integer mdebut;
22.   @Column(name = "HFIN", nullable = false)
23.   private Integer hfin;
24.   @Column(name = "MFIN", nullable = false)
25.   private Integer mfin;
26.
27.   // constructeur par défaut
28.   public Creneau() {
29.
30.   }
31.
32.   // constructeur avec paramètres
33.   public Creneau(Medecin medecin, Integer hDebut,Integer mDebut, Integer hFin, Integer mFin) {
34.     // on passe par les setters
35. ...
36.   }
37.
```

```
38.   // constructeur par recopie
39.   public Creneau(Creneau creneau) {
40.     // on passe par les setters
41. ...
42.   }
43.
44.   // toString
45.   @Override
46.   public String toString() {
47.     return "[" + getId() + "," + getVersion() + "," + getMedecin() + "," + getHdebut() + ":" +
    getMdebut() + "," + getHfin() + ":" + getMfin() + "]";
48.   }
49.
50.   // setters - getters
51. ...
52. }
```

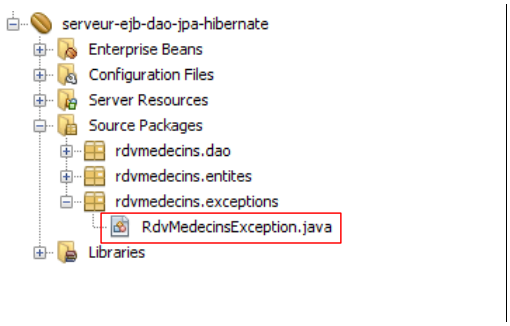
- les lignes 15-17 modélisent la relation "plusieurs à un" qui existe entre la table [creneaux] et la table [medecins] de la base de données : un médecin a plusieurs créneaux, un créneau appartient à un seul médecin.

L'entité [**Rv**] encapsule les lignes de la table [rv] :

```
1.  package rdvmedecins.entites;
2.  ...
3.  @Entity
4.  @Table(name = "RV")
5.  public class Rv implements Serializable {
6.    // caractéristiques
7.
8.    @Id
9.    @GeneratedValue(strategy = GenerationType.AUTO)
10.   @Column(name = "ID")
11.   private Long id;
12.   @Column(name = "JOUR", nullable = false)
13.   @Temporal(TemporalType.DATE)
14.   private Date jour;
15.   @ManyToOne
16.   @JoinColumn(name = "ID_CLIENT", nullable = false)
17.   private Client client;
18.   @ManyToOne
19.   @JoinColumn(name = "ID_CRENEAU", nullable = false)
20.   private Creneau creneau;
21.
22.   // constructeur par défaut
23.   public Rv() {
24.   }
25.
26.   // constructeur avec paramètres
27.   public Rv(Date jour, Client client, Creneau creneau) {
28.     // on passe par les setters
29. ...
30.   }
31.
32.   // constructeur par recopie
33.   public Rv(Rv rv) {
34.     // on passe par les setters
35. ...
36.   }
37.
38.   // toString
39.   @Override
40.   public String toString() {
41.     return "[" + getId() + "," + new SimpleDateFormat("dd/MM/yyyy").format(getJour()) + "," +
    getClient() + "," + getCreneau() + "]";
42.   }
43.
44. // getters et setters
45. ...
46. }
```

- les lignes 15-17 modélisent la relation "plusieurs à un" qui existe entre la table [rv] et la table [clients] (un client peut apparaître dans plusieurs Rv) de la base de données et les lignes 18-20 la relation "plusieurs à un" qui existe entre la table [rv] et la table [creneaux] (un créneau peut apparaître dans plusieurs Rv).

4.6.3 La classe d'exception

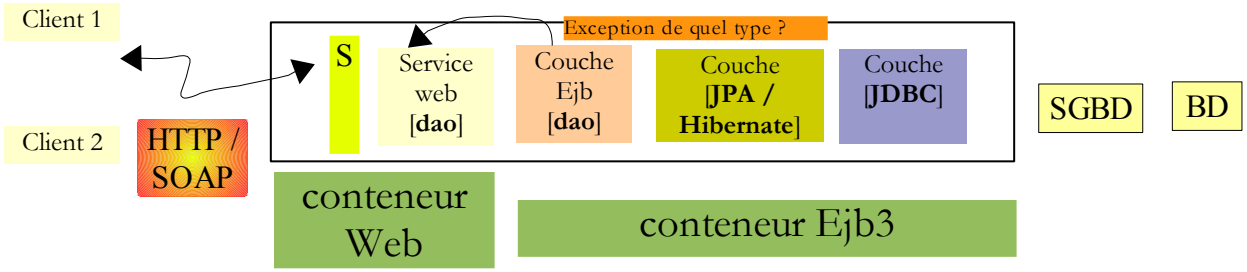


La classe d'exception [RdvMedecinsException] de l'application est la suivante :

```
1.  package rdvmedecins.exceptions;
2.
3.  import javax.ejb.ApplicationException;
4.
5.  @ApplicationException(rollback=true)
6.  public class RdvMedecinsException extends RuntimeException {
7.
8.    private static final long serialVersionUID = 1L;
9.
10.   // champs privés
11.   private int code = 0;
12.
13.   // constructeurs
14.   public RdvMedecinsException() {
15.     super();
16.   }
17.
18.   public RdvMedecinsException(String message) {
19.     super(message);
20.   }
21.
22.   public RdvMedecinsException(String message, Throwable cause) {
23.     super(message, cause);
24.   }
25.
26.   public RdvMedecinsException(Throwable cause) {
27.     super(cause);
28.   }
29.
30.   public RdvMedecinsException(String message, int code) {
31.     super(message);
32.     setCode(code);
33.   }
34.
35.   public RdvMedecinsException(Throwable cause, int code) {
36.     super(cause);
37.     setCode(code);
38.   }
39.
40.   public RdvMedecinsException(String message, Throwable cause, int code) {
41.     super(message, cause);
42.     setCode(code);
43.   }
44.
45.   // getters - setters
46. ...
47. }
```

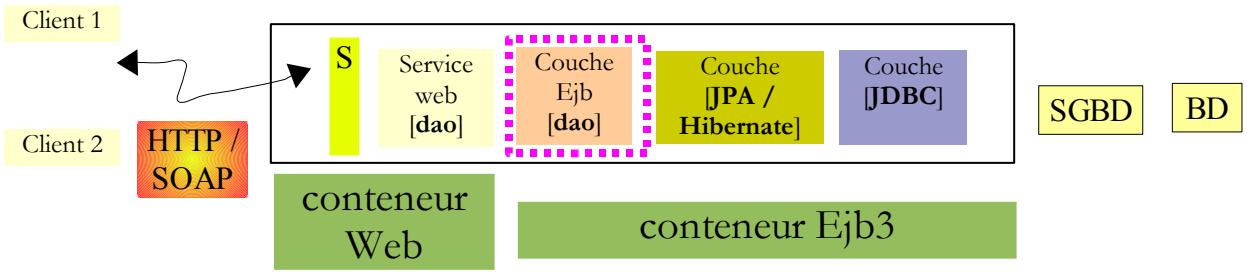
- ligne 6 : la classe dérive de la classe [RuntimeException]. Le compilateur ne force donc pas à la gérer avec des try / catch.
- ligne 5 : l'annotation **@ApplicationException** fait que l'exception ne sera pas "avalée" par une exception de type [EjbException].

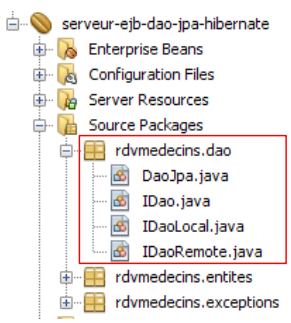
Pour comprendre l'annotation **@ApplicationException** revenons à l'architecture utilisée côté serveur :



L'exception de type [RdvMedecinsException] sera lancée par les méthodes de l'Ejb de la couche [dao] à l'intérieur du conteneur Ejb3 et interceptée par celui-ci. Sans l'annotation **@ApplicationException** le conteneur Ejb3 encapsule l'exception survenue, dans une exception de type [EjbException] et relance celle-ci. On peut ne pas vouloir de cette encapsulation et laisser sortir du conteneur Ejb3 une exception de type [RdvMedecinsException]. C'est ce que permet l'annotation **@ApplicationException**. Par ailleurs, l'attribut (**rollback=true**) de cette annotation indique au conteneur Ejb3 que si l'exception de type [RdvMedecinsException] se produit à l'intérieur d'une méthode exécutée au sein d'une transaction avec un SGBD, celle-ci doit être annulée. En termes techniques, cela s'appelle faire un *rollback* de la transaction.

4.6.4 L'Ejb de la couche [dao]





L'interface java [IDao] de la couche [dao] est la suivante :

```
1.  package rdvmedecins.dao;
2.  ...
3.  public interface IDao {
4.
5.    // liste des clients
6.    public List<Client> getAllClients();
7.    // liste des Médecins
8.    public List<Medecin> getAllMedecins();
9.    // liste des créneaux horaires d'un médecin
10.   public List<Creneau> getAllCreneaux(Medecin medecin);
11.   // liste des Rv d'un médecin, un jour donné
12.   public List<Rv> getRvMedecinJour(Medecin medecin, String jour);
13.   // trouver un client identifié par son id
14.   public Client getClientById(Long id);
15.   // trouver un client idenbtifié par son id
16.   public Medecin getMedecinById(Long id);
17.   // trouver un Rv identifié par son id
18.   public Rv getRvById(Long id);
19.   // trouver un créneau horaire identifié par son id
```

```
20.   public Creneau getCreneauById(Long id);
21.   // ajouter un RV
22.   public Rv ajouterRv(String jour, Creneau creneau, Client client);
23.   // supprimer un RV
24.   public void supprimerRv(Rv rv);
25. }
```

L'interface locale [IDaoLocal] de l'Ejb se contente de dériver l'interface [IDao] précédente :

```
1.  package rdvmedecins.dao;
2.
3.  import javax.ejb.Local;
4.
5.  @Local
6.  public interface IDaoLocal extends IDao{
7.  }
```

Il en est de même pour l'interface distante [IDaoRemote] :

```
1.  package rdvmedecins.dao;
2.
3.  import javax.ejb.Remote;
4.
5.  @Remote
6.  public interface IDaoRemote extends IDao {
7.  }
```

L'Ejb [DaoJpa] implémente les deux interfaces, **locale** et **distante** :

```
1.  package rdvmedecins.dao;
2.  ...
3.  @Stateless(mappedName="rdvmedecins.dao")
4.  @TransactionAttribute(TransactionAttributeType.REQUIRED)
5.  public class DaoJpa implements IDaoLocal,IDaoRemote {
6.  ...
7.  }
```

- la ligne 3 indique que l'Ejb distant porte le nom "rdvmedecins.dao"
- la ligne 4 indique que toutes les méthodes de l'Ejb se déroulent au sein d'une transaction gérée par le conteneur Ejb3.
- la ligne 5 montre que l'Ejb implémente les interfaces **locale** et **distante**.

Le code complet de l'Ejb est le suivant :

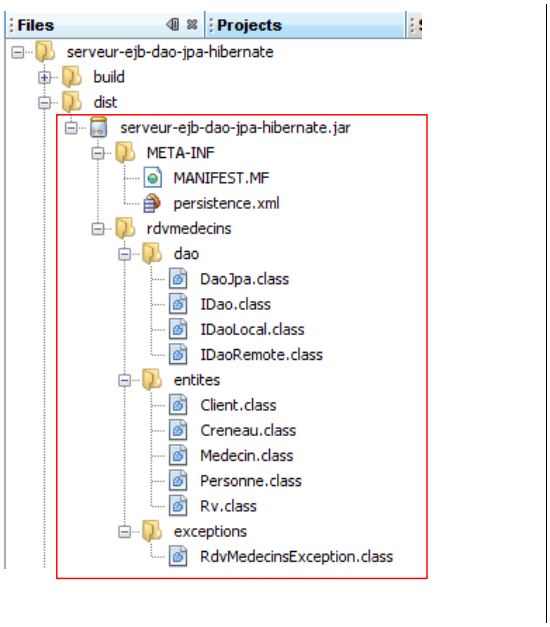
```
1.  package rdvmedecins.dao;
2.  ...
3.  @Stateless(mappedName="rdvmedecins.dao")
4.  @TransactionAttribute(TransactionAttributeType.REQUIRED)
5.  public class DaoJpa implements IDaoLocal,IDaoRemote {
6.
7.    @PersistenceContext
8.    private EntityManager em;
9.
10.   // liste des clients
11.   public List<Client> getAllClients() {
12.     try {
13.       return em.createQuery("select c from Client c").getResultList();
14.     } catch (Throwable th) {
15.       throw new RdvMedecinsException(th, 1);
16.     }
17.   }
18.
19.   // liste des médecins
20.   public List<Medecin> getAllMedecins() {
21.     try {
22.       return em.createQuery("select m from Medecin m").getResultList();
23.     } catch (Throwable th) {
24.       throw new RdvMedecinsException(th, 2);
25.     }
26.   }
27.
28.   // liste des créneaux horaires d'un médecin donné
29.   // medecin : le médecin
30.   public List<Creneau> getAllCreneaux(Medecin medecin) {
31.     try {
```

```
32.       return em.createQuery("select c from Creneau c join c.medecin m where
    m.id=:idMedecin").setParameter("idMedecin", medecin.getId()).getResultList();
33.     } catch (Throwable th) {
34.       throw new RdvMedecinsException(th, 3);
35.     }
36.   }
37.
38.   // liste des Rv d'un médecin donné, un jour donné
39.   // medecin : le médecin
40.   // jour : le jour
41.   public List<Rv> getRvMedecinJour(Medecin medecin, String jour) {
42.     try {
43.       return em.createQuery("select rv from Rv rv join rv.creneau c join c.medecin m where
    m.id=:idMedecin and rv.jour=:jour").setParameter("idMedecin",
    medecin.getId()).setParameter("jour", new
    SimpleDateFormat("yyyy:MM:dd").parse(jour)).getResultList();
44.     } catch (Throwable th) {
45.       throw new RdvMedecinsException(th, 4);
46.     }
47.   }
48.
49.   // ajout d'un Rv
50.   // jour : jour du Rv
51.   // creneau : créneau horaire du Rv
52.   // client : client pour lequel est pris le Rv
53.   public Rv ajouterRv(String jour, Creneau creneau, Client client) {
54.     try {
55.       Rv rv = new Rv(new SimpleDateFormat("yyyy:MM:dd").parse(jour), client, creneau);
56.       em.persist(rv);
57.       return rv;
58.     } catch (Throwable th) {
59.       throw new RdvMedecinsException(th, 5);
60.     }
61.   }
62.
63.   // suppression d'un Rv
64.   // rv : le Rv supprimé
65.   public void supprimerRv(Rv rv) {
66.     try {
67.       em.remove(em.merge(rv));
68.     } catch (Throwable th) {
69.       throw new RdvMedecinsException(th, 6);
70.     }
71.   }
72.
73.   // récupérer un client donné
74.   public Client getClientById(Long id) {
75.     try {
76.       return (Client) em.find(Client.class, id);
77.     } catch (Throwable th) {
78.       throw new RdvMedecinsException(th, 7);
79.     }
80.   }
81.
82.   // récupérer un médecin donné
83.   public Medecin getMedecinById(Long id) {
84.     try {
85.       return (Medecin) em.find(Medecin.class, id);
86.     } catch (Throwable th) {
87.       throw new RdvMedecinsException(th, 8);
88.     }
89.   }
90.
91.   // récupérer un Rv donné
92.   public Rv getRvById(Long id) {
93.     try {
94.       return (Rv) em.find(Rv.class, id);
95.     } catch (Throwable th) {
96.       throw new RdvMedecinsException(th, 9);
97.     }
98.   }
99.
100.    // récupérer un créneau donné
101.    public Creneau getCreneauById(Long id) {
102.      try {
103.        return (Creneau) em.find(Creneau.class, id);
104.      } catch (Throwable th) {
105.        throw new RdvMedecinsException(th, 10);
```

```
106.       }
107.     }
108.   }
```

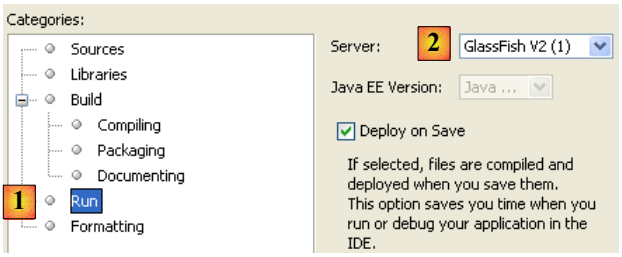
- ligne 8 : l'objet **EntityManager** qui gère l'accès au contexte de persistance. A l'instanciation de la classe, ce champ sera initialisé par le conteneur Ejb grâce à l'annotation **@PersistenceContext** de la ligne 7.
- ligne 15 : requête JPQL qui retourne toutes les lignes de la table [clients] sous la forme d'une liste d'objets [Client].
- ligne 22 : requête analogue pour les médecins
- ligne 32 : une requête JPQL réalisant une jointure entre les tables [creneaux] et [medecins]. Elle est paramétrée par l'id du médecin.
- ligne 43 : une requête JPQL réalisant une jointure entre les tables [rv], [creneaux] et [medecins] et ayant deux paramètres : l'id du médecin et le jour du Rv.
- lignes 55-57 : création d'un Rv puis persistance de celui-ci en base de données.
- ligne 67 : suppression d'un Rv en base de données.
- ligne 76 : réalise un select sur la base de données pour trouver un client donné
- ligne 85 : idem pour un médecin
- ligne 94 : idem pour un Rv
- ligne 103 : idem pour un créneau horaire
- toutes les opérations avec le contexte de persistance **em** de la ligne 9 sont susceptibles de rencontrer un problème avec la base de données. Aussi sont-elles toutes entourées par un try / catch. L'éventuelle exception est encapsulée dans l'exception "maison" **RdvMedecinsException**.

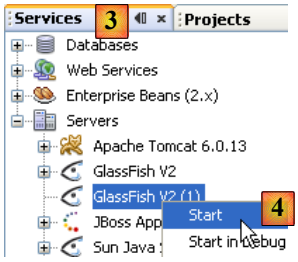
Le module Ejb une fois compilé donne naissance à un fichier .jar :



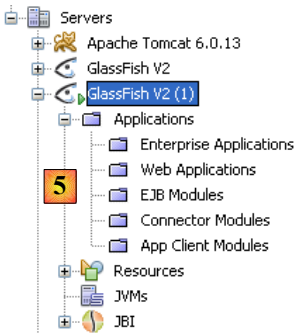
4.7 Déploiement de l'Ejb de la couche [dao] avec Netbeans

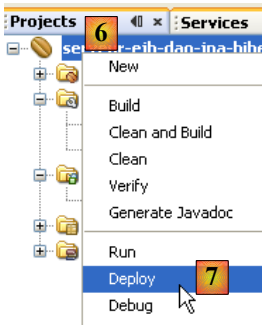
Netbeans permet de déployer de façon simple sur le serveur Glassfish l'Ejb créé précédemment.

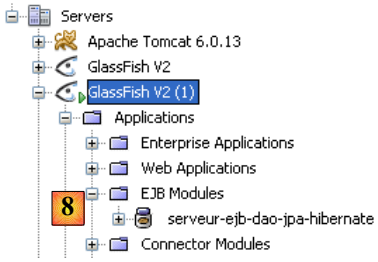




- dans les propriétés du projet Ejb, vérifier les options d'exécution [1].
- en [2], le nom du serveur sur lequel va être déployé l'Ejb
- dans l'onglet [Services] [3], on le lance [4].



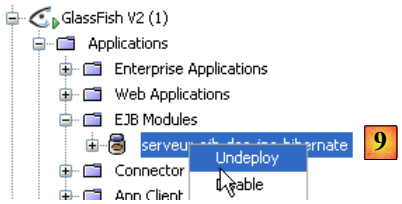


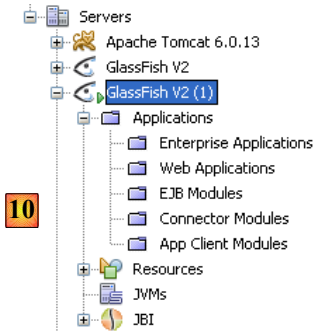


- en [5], le serveur Glassfish une fois lancé. Il n'a pas encore de module Ejb.

- lancez le serveur MySQL et assurez-vous que la base [dbrdvmedecins] est en ligne. Pour cela, vous pouvez utiliser la connexion Netbeans créée au paragraphe 4.5, page 11.

- dans l'onglet [Projects] [6], on déploie le module Ejb [7] : il faut que le SGBD MySQL5 soit lancé pour que la ressource JDBC "jdbc/dbrdvmedecins" utilisée par l'Ejb soit accessible.
- en [8], l'Ejb déployé apparaît dans l'arborescence du serveur Glassfish





- en [9], on enlève l'Ejb déployé
- en [10], l'Ejb n'apparaît plus dans l'arborescence du serveur Glassfish.

4.8 Déploiement de l'Ejb de la couche [dao] avec Glassfish

Nous montrons ici comment déployer sur le serveur Glassfish un Ejb à partir de son archive .jar.

- lancez le serveur MySQL et assurez-vous que la base [dbrdvmedecins] est en ligne. Pour cela, vous pouvez utiliser la connexion Netbeans créée au paragraphe 4.5, page 11.

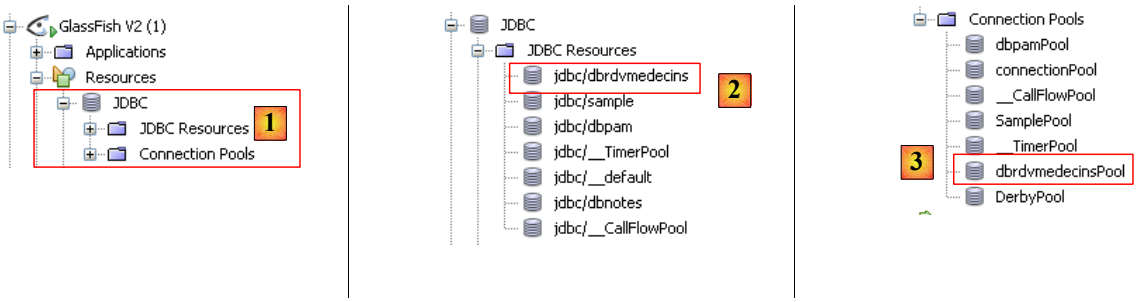
Rappelons la configuration Jpa du module Ejb qui va être déployé. Cette configuration est faite dans le fichier [persistence.xml] :

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <persistence version="1.0" 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_1_0.xsd">
3.    <persistence-unit name="dbrdvmedecins" transaction-type="JTA">
4.      <provider>org.hibernate.ejb.HibernatePersistence</provider>
5.      <jta-data-source>jdbc/dbrdvmedecins</jta-data-source>
6.      <properties>
7.        <!-- Dialecte -->
8.        <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5InnoDBDialect"/>
9.      </properties>
10.   </persistence-unit>
11. </persistence>
```

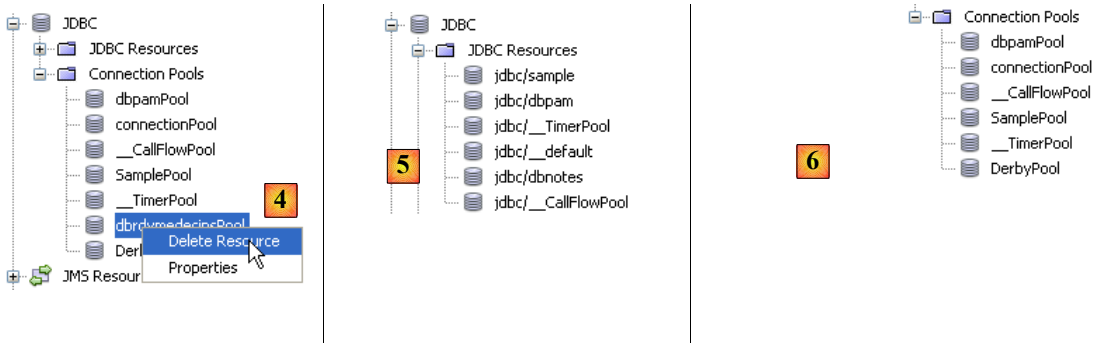
La ligne 5 indique que la couche Jpa utilise une source de données JTA, c.a.d. gérée par le conteneur Ejb3, nommée "jdbc/dbrdvmedecins".

Nous avons vu au paragraphe 4.5, page 13, comment créer cette ressource JDBC à partir de Netbeans. Nous montrons ici comment faire la même chose directement avec Glassfish. Nous suivons ici une procédure décrite au paragraphe 13.1.2, page 79 de [**ref1**].

Nous commençons par supprimer la ressource afin de pouvoir la recréer. Nous le faisons à partir de Netbeans :

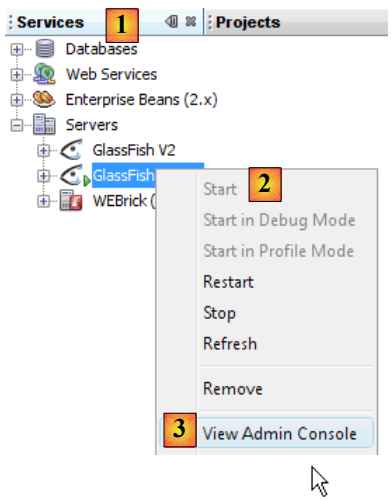


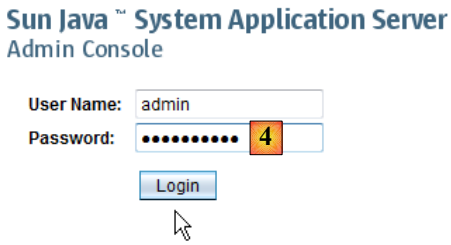
- en [1], les ressources JDBC du serveur Glassfish
- en [2], la ressource "jdbc/dbrdvmedecins" de notre Ejb
- en [3], le pool de connexions de cette ressource JDBC



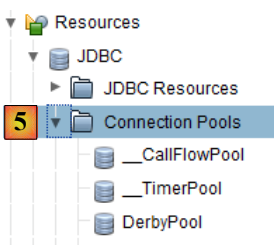
- en [4], on supprime le pool de connexions. Cela va avoir pour effet de supprimer toutes les ressources JDBC qui l'utilisent, donc la ressource "jdbc/dbrdvmedecins".
- en [5] et [6], la ressource JDBC et le pool de connexions ont été détruits.

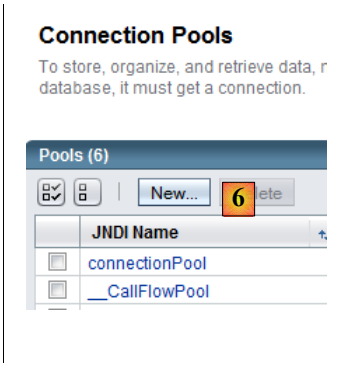
Maintenant, nous utilisons la console d'administration du serveur Glassfish pour créer la ressource JDBC et déployer l'Ejb.

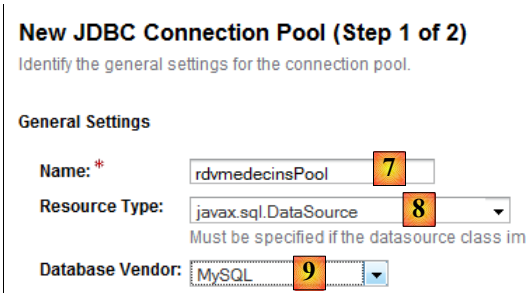




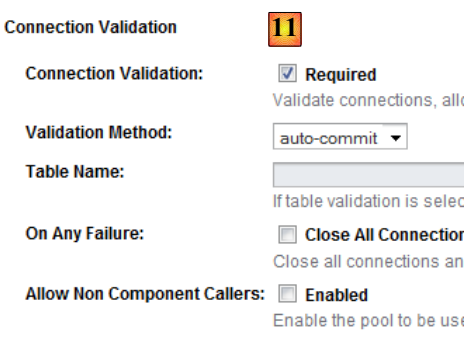
- dans l'onglet [services] [1] de Netbeans, lancez le serveur Glassfish [2] puis accédez [3] à sa console d'administration
- en [4], connectez-vous comme administrateur (mot de passe : *adminadmin* si vous n'avez pas changé celui-ci lors de l'installation ou après).

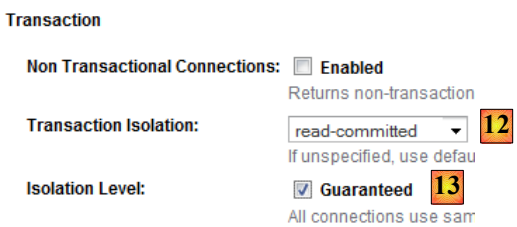




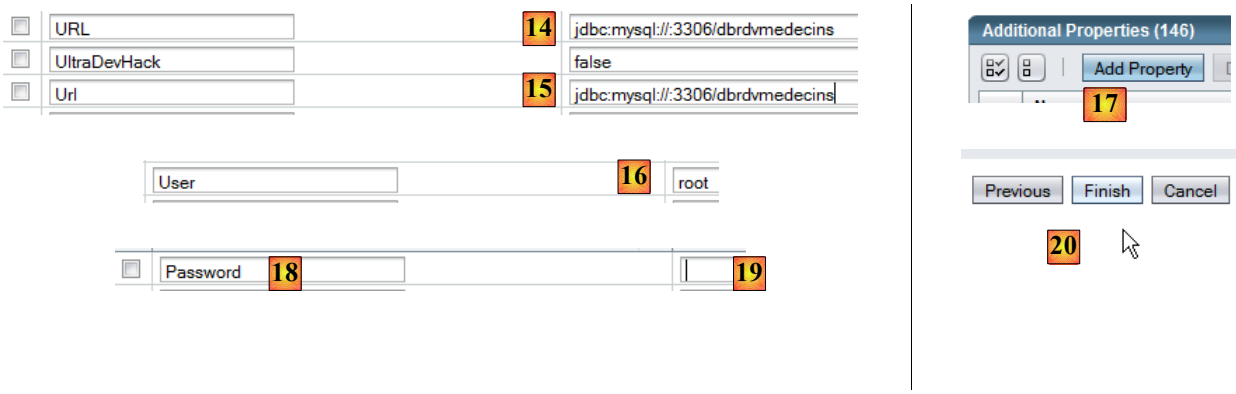


- en [5], sélectionnez la branche [Connection Pools] des ressources de Glassfish
- en [6], créez un nouveau pool de connexions. On rappelle qu'un pool de connexions est une technique pour limiter le nombre d'ouvertures / fermetures de connexions avec un SGBD. Au démarrage du serveur, N, un nombre défini par configuration, connexions sont ouvertes avec le SGBD. Ces connexions ouvertes sont ensuites mises à disposition des Ejb qui les sollicitent pour faire une opération avec le SGBD. Dès que celle-ci est terminée, l'Ejb rend la connexion au pool. La connexion n'est jamais fermée. Elle est partagée entre les différents threads qui accèdent au SGBD
- en [7], donnez un nom au pool
- en [8], la classe modélisant la source de données est la classe [javax.sql.DataSource]
- en [9], le SGBD qui détient la source de données est ici MySQl.
- en [10], passez à l'étape suivante

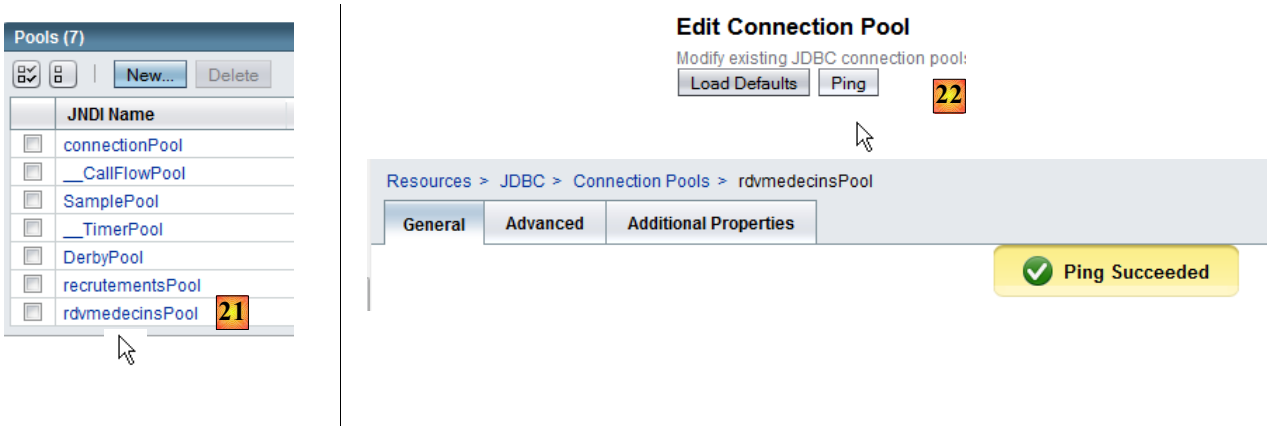




- en [11], l'attribut "Connection Validation Required" fait qu'avant de donner une connexion, le pool vérifie qu'elle est opérationnelle. Si ce n'est pas le cas, il en crée une nouvelle. Ceci permet à une application de continuer à fonctionner après une coupure momentanée avec le SGBD. Pendant la coupure, aucune connexion n'est utilisable et des exceptions sont remontées au client. Lorsque la coupure est terminée, les clients qui continuent à demander des connexions les obtiennent de nouveau : grâce à l'attribut "Connection Validation Required" toutes les connexions du pool vont être recréées. Sans cet attribut, le pool constaterait que les connexions initiales ont été coupées mais ne chercherait pas à en recréer de nouvelles.
- en [12], on demande le niveau d'isolement "Read Committed" pour les transactions. Ce niveau assure que une transaction T2 ne peut lire des données modifiées par une transaction T1 tant que cette dernière n'est pas totalement terminée.
- en [13], on demande à ce que toutes les transactions utilisent le niveau d'isolement précisé en [12]



- en [14] et [15], précisez l'Url de la BD dont le pool gère les connexions
- en [16], l'utilisateur sera *root*
- en [17], ajoutez une propriété
- en [18], ajoutez la propriété "Password" avec la valeur () en [19]. Bien que la copie d'écran [19] ne le montre pas, il ne faut pas mettre la chaîne vide mais bien **()** (parenthèse ouvrante, parenthèse fermante) pour désigner un mot de passe vide. Si l'utilisateur *root* de votre SGBD MySQL a un mot de passe non vide, mettez ce mot de passe.
- en [20], terminez l'assistant de création du pool de connexions pour la base MySQL [dbrdvmedecins].



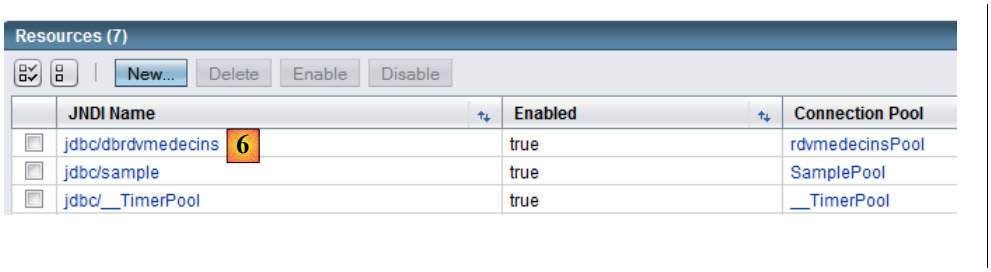
- en [21], le pool a été créé. On clique sur son lien.
- en [22], le boton [Ping] permet de créer une connexion avec la bd [dbrdvmedecins]
- en [23], si tout va bien, un message indique que la connexion a réussi

Une fois le pool de connexions créé, on peut créer une ressource Jdbc :

- en [1], on sélectionne la branche [JDBC Resources] de l'arbre des objets du serveur
- en [2], on crée une nouvelle ressource JDBC
- en [3], on donne un nom à la ressource JDBC. Celui-ci doit correspondre au nom utilisé dans le fichier [persistence.xml] :

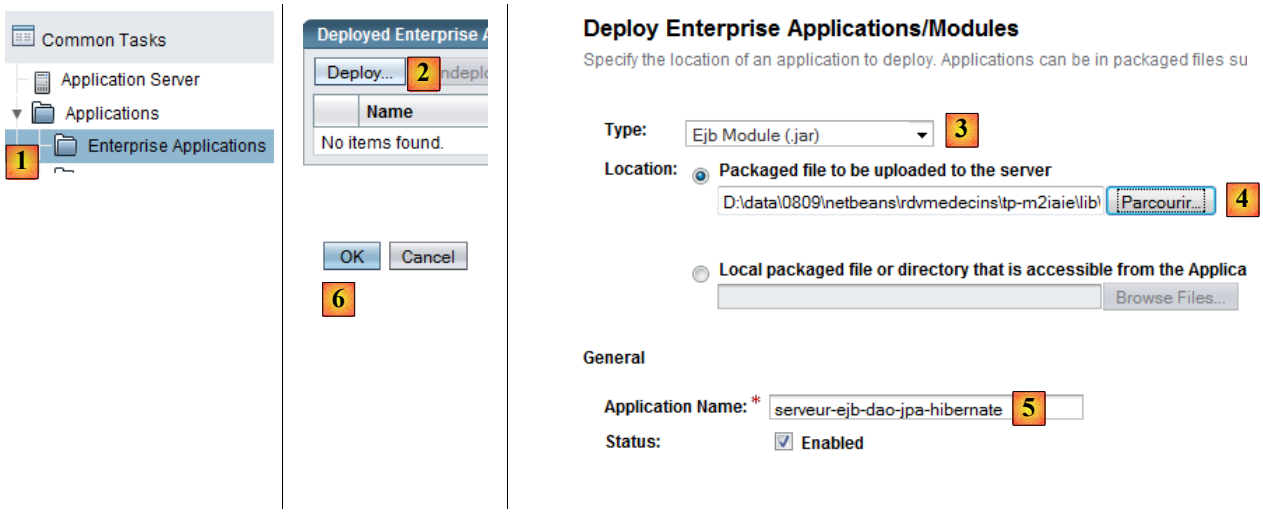
```
<jta-data-source>jdbc/dbrdvmedecins</jta-data-source>
```

- en [4], on précise le pool de connexions que doit utiliser la nouvelle ressource JDBC : celui qu'on vient de créer
- en [5], on termine l'assistant de création

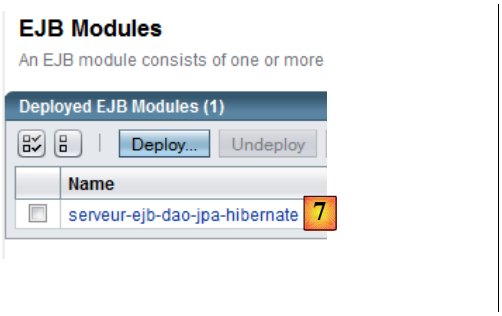


- en [6] la nouvelle ressource JDBC

Maintenant que la ressource JDBC est créée, on peut déployer l'archive jar de l'Ejb :



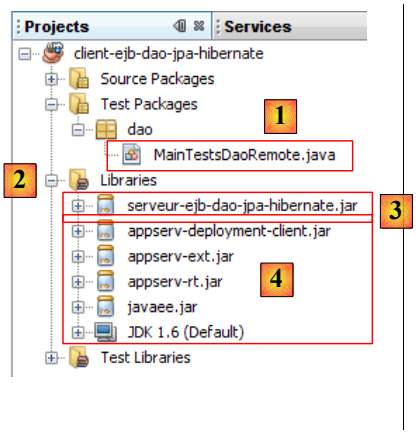
- en [1], sélectionnez la branche [Enterprise Applications]
- en [2], avec le bouton [Deploy], indiquez que vous voulez déployer une nouvelle application
- en [3], indiquez que l'application est un module Ejb
- en [4], sélectionnez le jar de l'Ejb [serveur-ejb-dao-jpa-hibernate.jar] qui vous aura été donné pour le TP.
- en [5], vous pouvez changer le nom du module Ejb si vous le souhaitez
- en [6], terminez l'assistant de déploiement du module ejb



- en [7], le module Ejb a été déployé. Il peut désormais être utilisé.

4.9 Tests de l'Ejb de la couche [dao]

Maintenant que l'Ejb de la couche [dao] de notre application a été déployé, nous pouvons le tester. Nous le ferons au moyen du client Java suivant :



La classe [MainTestsDaoRemote] [1] est une classe de test JUnit 4. Les bibliothèques en [2] sont constituées d'une part :

- du jar de l'Ejb de la couche [dao] [3] (cf page 30).
- des bibliothèques Glassfish [4] nécessaires aux clients distants des Ejb.

La classe de test est la suivante :

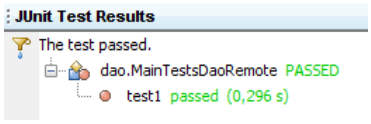
```
1.  package dao;
2.  ...
3.  public class MainTestsDaoRemote {
4.
5.    // couche [dao] testée
6.    private static IDaoRemote dao;
7.
8.    @BeforeClass
9.    public static void init() throws NamingException {
10.     // initialisation environnement JNDI
11.     InitialContext initialContext = new InitialContext();
12.     // instanciation couche dao
13.     dao = (IDaoRemote) initialContext.lookup("rdvmedecins.dao");
14.   }
15.
16.   @Test
17.   public void test1() {
18.     // données du test
19.     String jour = "2006:08:23";
20.     // affichage clients
21.     List<Client> clients = null;
```

```
22.     try {
23.       clients = dao.getAllClients();
24.       display("Liste des clients :", clients);
25.     } catch (Exception ex) {
26.       System.out.println(ex);
27.     }
28.     // affichage médecins
29.     List<Medecin> medecins = null;
30.     try {
31.       medecins = dao.getAllMedecins();
32.       display("Liste des médecins :", medecins);
33.     } catch (Exception ex) {
34.       System.out.println(ex);
35.     }
36.     // affichage créneaux d'un médecin
37.     Medecin medecin = medecins.get(0);
38.     List<Creneau> creneaux = null;
39.     try {
40.       creneaux = dao.getAllCreneaux(medecin);
41.       display(String.format("Liste des créneaux du médecin %s", medecin), creneaux);
42.     } catch (Exception ex) {
43.       System.out.println(ex);
44.     }
45.     // liste des Rv d'un médecin, un jour donné
46.     try {
47.       display(String.format("Liste des créneaux du médecin %s, le [%s]", medecin, jour),
    dao.getRvMedecinJour(medecin, jour));
48.     } catch (Exception ex) {
49.       System.out.println(ex);
50.     }
51.     // ajouter un RV
52.     Rv rv = null;
53.     Creneau creneau = creneaux.get(2);
54.     Client client = clients.get(0);
55.     System.out.println(String.format("Ajout d'un Rv le [%s] dans le créneau %s pour le client %s",
    jour, creneau, client));
56.     try {
57.       rv = dao.ajouterRv(jour, creneau, client);
58.       System.out.println("Rv ajouté");
59.       display(String.format("Liste des Rv du médecin %s, le [%s]", medecin, jour),
    dao.getRvMedecinJour(medecin, "2006:08:23"));
60.     } catch (Exception ex) {
61.       System.out.println(ex);
62.     }
63.     // ajouter un RV dans le même créneau du même jour
64.     // doit provoquer une exception
65.     System.out.println(String.format("Ajout d'un Rv le [%s] dans le créneau %s pour le client %s",
    jour, creneau, client));
66.     try {
67.       rv = dao.ajouterRv(jour, creneau, client);
68.       System.out.println("Rv ajouté");
69.       display(String.format("Liste des Rv du médecin %s, le [%s]", medecin, jour),
    dao.getRvMedecinJour(medecin, "2006:08:23"));
70.     } catch (Exception ex) {
71.       System.out.println(ex);
72.     }
73.     // supprimer un RV
74.     System.out.println("Suppression du Rv ajouté");
75.     try {
76.       dao.supprimerRv(rv);
77.       System.out.println("Rv supprimé");
78.       display(String.format("Liste des Rv du médecin %s, le [%s]", medecin, jour),
    dao.getRvMedecinJour(medecin, "2006:08:23"));
79.     } catch (Exception ex) {
80.       System.out.println(ex);
81.     }
82.   }
83.
84.   // méthode utilitaire - affiche les éléments d'une collection
85.   private static void display(String message, List elements) {
86.     System.out.println(message);
87.     for (Object element : elements) {
88.       System.out.println(element);
89.     }
90.   }
91. }
```

- ligne 13 : on notera l'instanciation du proxy de l'Ejb distant. On utilise son nom JNDI "**rdvmedecins.dao**".

- les méthodes de test utilisent les méthodes exposées par l'Ejb (cf page 27).

Si tout va bien, les tests doivent passer :

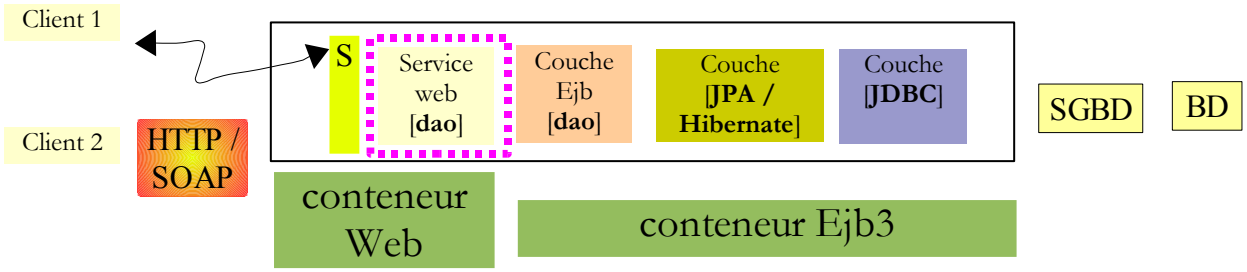


Maintenant que l'Ejb de la couche [dao] est opérationnel, on peut passer à son exposition publique via un service web.

4.10 Le service web de la couche [dao]

Pour une courte introduction à la notion de service web, on lira le paragraphe 14, page 111 de [**ref1**].

Revenons à l'architecture du serveur de notre application client / serveur :

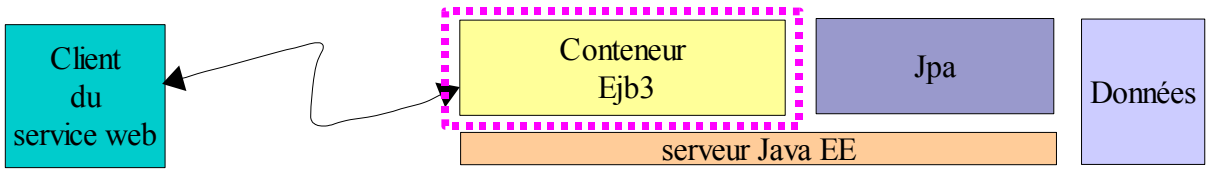


Nous nous intéressons ci-dessus au service web de la couche [dao]. Ce service a pour seul rôle de rendre disponible l'interface de l'Ejb de la couche [dao] à des clients multi-plateformes capables de dialoguer avec un service web.

Rappelons qu'il y a deux façons d'implémenter un service web :

- par une classe annotée **@WebService** qui s'exécute dans un conteneur web

- par un Ejb annoté **@WebService** qui s'exécute dans un conteneur Ejb



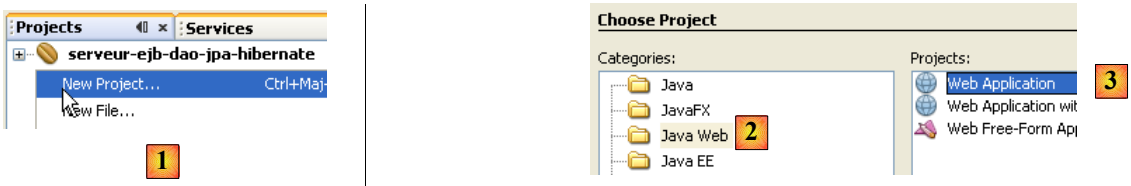
Nous utilisons ici la première solution. Dans l'IDE Netbeans, il nous faut construire un projet d'entreprise avec deux modules :

- le module Ejb qui s'exécutera dans le conteneur Ejb : l'Ejb de la couche [dao].
- le module web qui s'exécutera dans le conteneur web : le service web que nous sommes en train de construire.

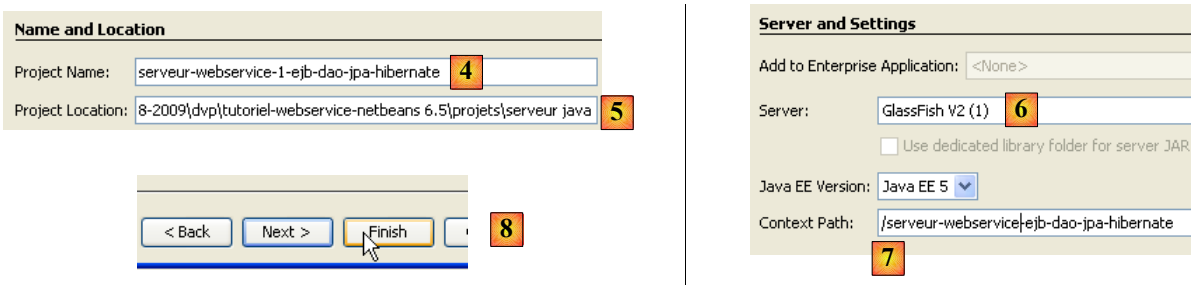
Nous allons construire ce projet d'entreprise de deux façons.

4.10.1 Projet Netbeans - Version 1

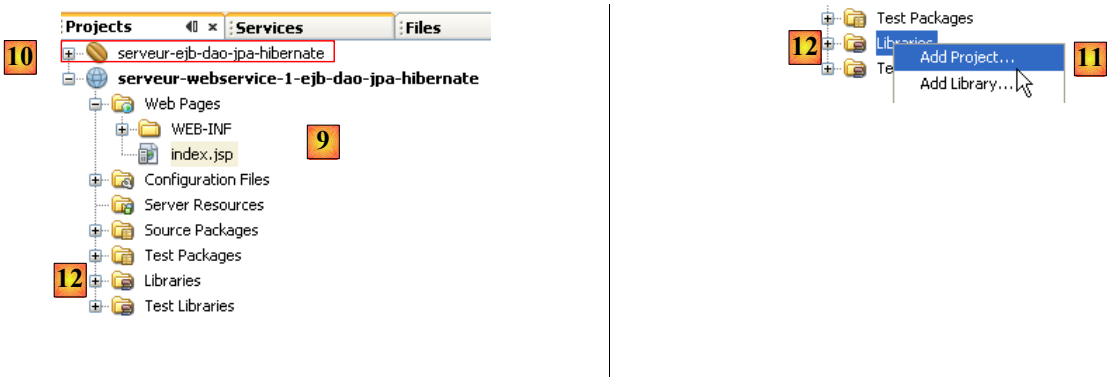
Nous construisons tout d'abord un projet Netbeans de type "Web Application" :



- en [1], on crée un nouveau projet dans la catégorie "Java Web" [2] de type "Web Application" [3].



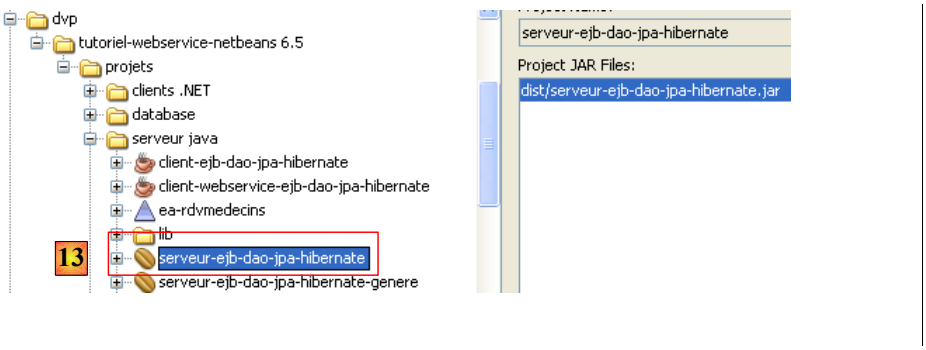
- en [4], on donne un nom au projet et en [5] on précise le dossier dans lequel il doit être généré
- en [6], on fixe le serveur d'application qui va exécuter l'application web
- en [7], on fixe le contexte de l'application
- en [8], on valide la configuration du projet.



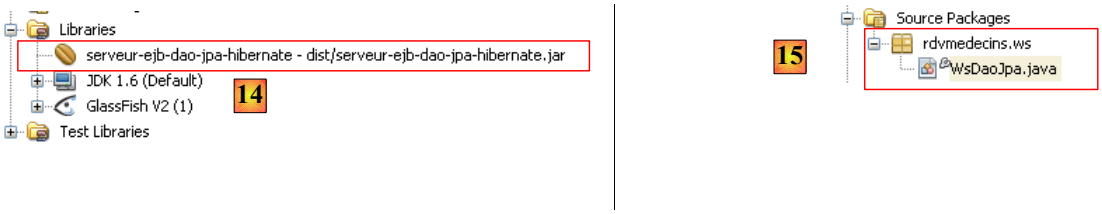
- en [9], le projet généré.

Le service web que nous construisons va utiliser l'EJB du projet précédent [10]. Aussi a-t-il besoin de référencer le .jar du module Ejb [10].

- en [11], on ajoute un projet Netbeans aux bibliothèques du projet web [12]



- en [13], on sélectionne le dossier du module Ejb dans le système de fichiers et on valide.



- en [14], le module Ejb a été ajouté aux bibliothèques du projet web.

En [15], nous implémentons le service web avec la classe [WsDaoJpa] suivante :

```
1.  package rdvmedecins.ws;
2.  ...
3.  @WebService()
4.  public class WsDaoJpa implements IDao {
5.
6.    @EJB
7.    private IDaoLocal dao;
8.
9.    // liste des clients
10.   @WebMethod
11.   public List<Client> getAllClients() {
12.     return dao.getAllClients();
13.   }
14.
15.   // liste des médecins
16.   @WebMethod
17.   public List<Medecin> getAllMedecins() {
18.     return dao.getAllMedecins();
19.   }
20.
21.   // liste des créneaux horaires d'un médecin donné
22.   // medecin : le médecin
23.   @WebMethod
24.   public List<Creneau> getAllCreneaux(Medecin medecin) {
25.     return dao.getAllCreneaux(medecin);
26.   }
27.
28.   // liste des Rv d'un médecin donné, un jour donné
29.   // medecin : le médecin
30.   // jour : le jour
31.   @WebMethod
32.   public List<Rv> getRvMedecinJour(Medecin medecin, String jour) {
33.     return dao.getRvMedecinJour(medecin, jour);
34.   }
35.
36.   // ajout d'un Rv
37.   // jour : jour du Rv
38.   // creneau : créneau horaire du Rv
39.   // client : client pour lequel est pris le Rv
40.   @WebMethod
41.   public Rv ajouterRv(String jour, Creneau creneau, Client client) {
42.     return dao.ajouterRv(jour, creneau, client);
43.   }
```

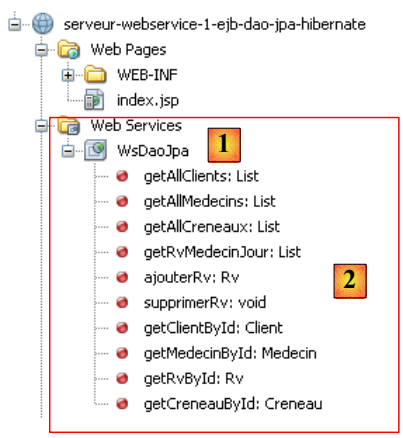
```
44.
45.   // suppression d'un Rv
46.   // rv : le Rv supprimé
47.   @WebMethod
48.   public void supprimerRv(Rv rv) {
49.     dao.supprimerRv(rv);
50.   }
51.
52.   // récupérer un client donné
53.   @WebMethod
54.   public Client getClientById(Long id) {
55.     return dao.getClientById(id);
56.   }
57.
58.   // récupérer un médecin donné
59.   @WebMethod
60.   public Medecin getMedecinById(Long id) {
61.     return dao.getMedecinById(id);
62.   }
63.
64.   // récupérer un Rv donné
65.   @WebMethod
66.   public Rv getRvById(Long id) {
67.     return dao.getRvById(id);
68.   }
69.
70.   // récupérer un créneau donné
71.   @WebMethod
72.   public Creneau getCreneauById(Long id) {
73.     return dao.getCreneauById(id);
74.   }
75. }
```

- ligne 4, la classe [WsdaoJpa] implémente l'interface [IDao]. Rappelons que cette interface est définie dans l'archive de l'Ejb de la couche [dao] sous la forme suivante :

```
a)  package rdvmedecins.dao;
b)  ...
c)  public interface IDao {
d)
e)    // liste des clients
f)    public List<Client> getAllClients();
g)    // liste des Médecins
h)    public List<Medecin> getAllMedecins();
i)    // liste des créneaux horaires d'un médecin
j)    public List<Creneau> getAllCreneaux(Medecin medecin);
k)    // liste des Rv d'un médecin, un jour donné
l)    public List<Rv> getRvMedecinJour(Medecin medecin, String jour);
m)    // trouver un client identifié par son id
n)    public Client getClientById(Long id);
o)    // trouver un client idenbtifié par son id
p)    public Medecin getMedecinById(Long id);
q)    // trouver un Rv identifié par son id
r)    public Rv getRvById(Long id);
s)    // trouver un créneau horaire identifié par son id
t)    public Creneau getCreneauById(Long id);
u)    // ajouter un RV
v)    public Rv ajouterRv(String jour, Creneau creneau, Client client);
w)    // supprimer un RV
x)    public void supprimerRv(Rv rv);
y)  }
```

- ligne 3 : l'annotation **@WebService** fait de la classe [WsDaoJpa] un service web.
- lignes 6-7 : la référence de l'Ejb de la couche [dao] sera injectée par le serveur d'applications dans le champ de la ligne 7. C'est l'implémentation locale *IDaoLocal* qui est ici injectée parce que le service web s'exécute dans la même Jvm que l'Ejb.
- toutes les méthodes du service web sont taguées avec l'annotation **@WebMethod** pour en faire des méthodes visibles aux clients distants. Une méthode non taguée avec l'annotation **@WebMethod** serait interne au service web et non visible aux clients distants. Chaque méthode M du service web se contente d'appeler la méthode M correspondante de l'Ejb injecté en ligne 7.

La création de ce service web est reflétée par une nouvelle branche dans le projet Netbeans :



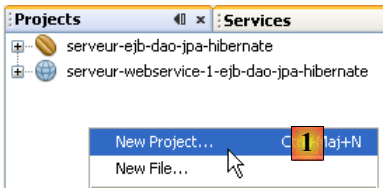
On voit en [1] le service web WsDaoJpa et en [2] les méthodes qu'il expose aux clients distants.

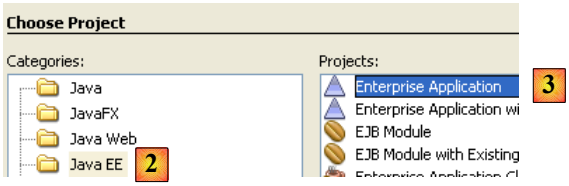
Rappelons l'architecture du service web en construction :

Les composantes du service web que nous allons déployer sont :

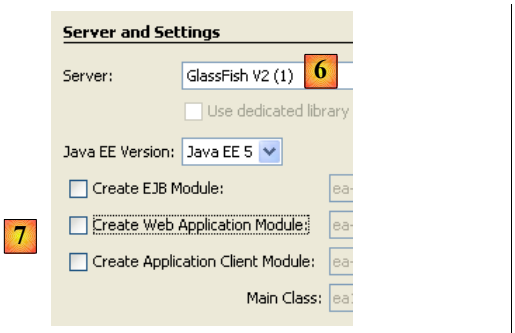
- [1] : le module web que nous venons de construire
- [2] : le module Ejb que nous avons construit lors d'une étape précédente et dont dépend le service web

Pour les déployer ensemble, il faut rassembler les deux modules dans un projet Netbeans dit "d'entreprise" :



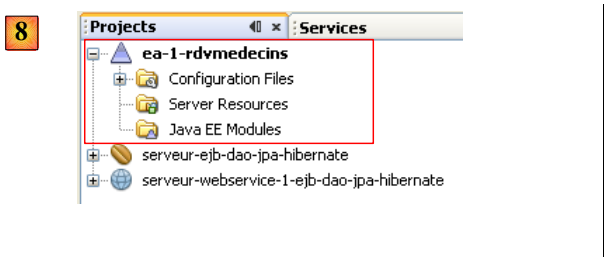


En [1] on crée un nouveau projet d'entreprise [2, 3].

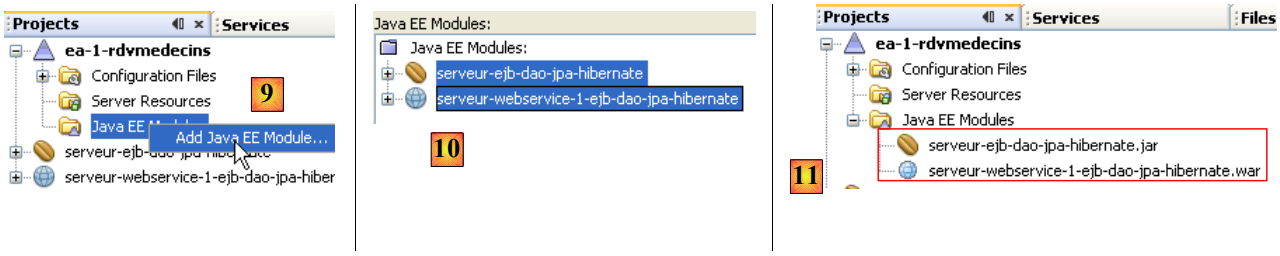


- en [4,5], on donne un nom au projet et on fixe son dossier de création

- en [6], on choisit le serveur d'application sur lequel sera déployée l'application d'entreprise
- en [7], un projet d'entreprise peut avoir trois composantes : application web, module Ejb, application cliente. Ici, le projet est créé sans aucune composante. Celles-ci seront rajoutées ultérieurement.

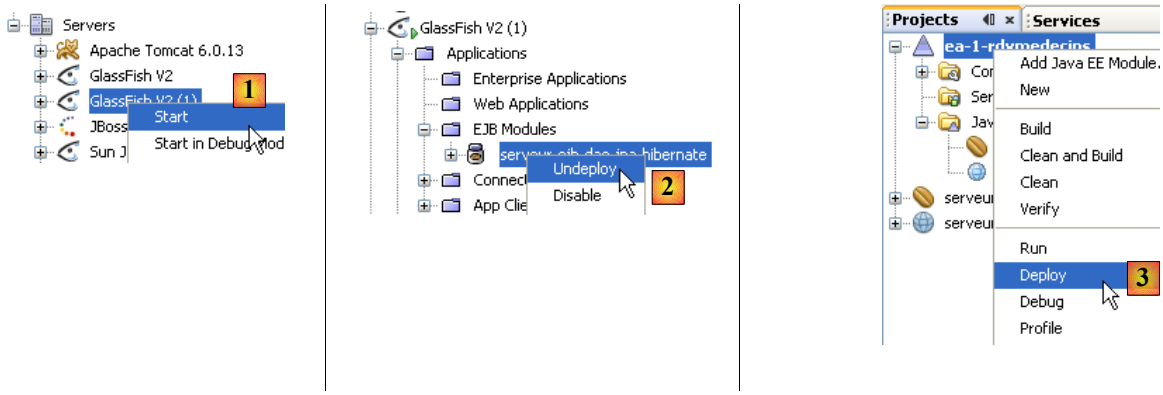


- en [8], l'application d'entreprise nouvellement créée.

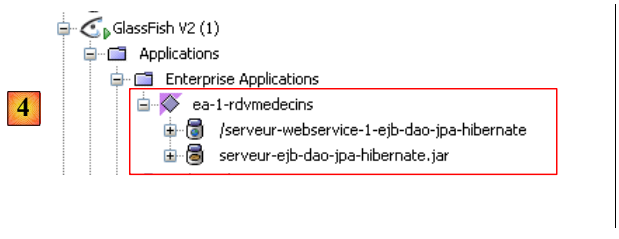


- en [9], cliquer droit sur [Java EE Modules] et ajouter un nouveau module
- en [10], seuls les modules Netbeans actuellement ouverts dans l'IDE sont présentés. Ici nous sélectionnons le module web [serveur-webservice-1-ejb-dao-jpa-hibernate] et le module Ejb [serveur-ejb-dao-jpa-hibernate] que nous avons construits.
- en [11], les deux modules ajoutés au projet d'entreprise.

Il nous reste à déployer cette application d'entreprise sur le serveur Glassfish. Pour la suite, le SGBD MySQL doit être lancé afin que la source de données JDBC "jdbc/dbrdvmedecins" utilisée par le module Ejb soit accessible.



- en [1], on lance le serveur Glassfish
- si le module Ejb [serveur-ejb-dao-jpa-hibernate] est déployé, on le décharge [2]
- en [3], on déploie l'application d'entreprise

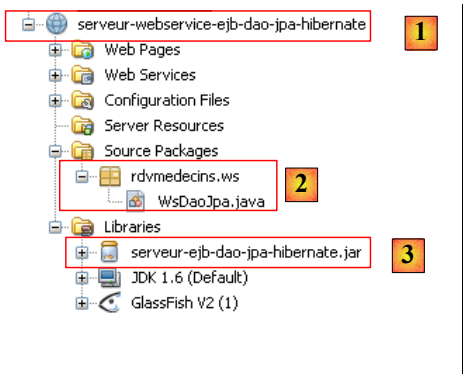


- en [4], elle est déployée. On voit qu'elle contient les deux modules : Web et Ejb.

4.10.2 Projet Netbeans - version 2

Nous montrons maintenant comment déployer le service web lorsqu'on ne dispose pas du code source du module Ejb mais seulement son archive .jar.

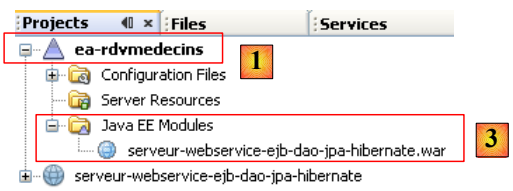
Le nouveau projet Netbeans du service web sera le suivant :



Les éléments notables du projet sont les suivants :

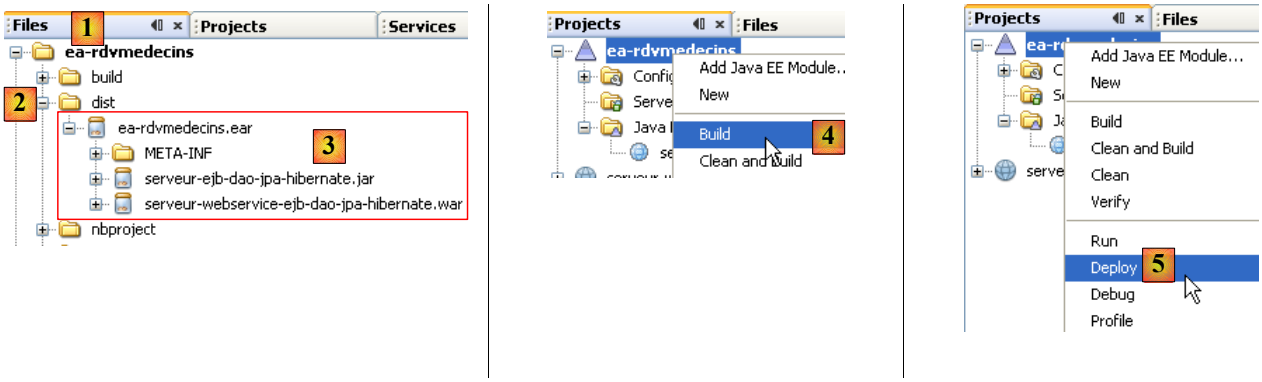
- [1] : le service web est implémenté par un projet Netbeans de type [Web Application].
- [2] : le service web est implémenté par la classe [WsDaoJpa] déjà étudiée
- [3] : l'archive de l'Ejb de la couche [dao] qui permet à la classe [WsDaoJpa] d'avoir accès aux définitions des différentes classes, interfaces, entités des couches [dao] et [jpa].

Nous construisons ensuite le projet d'entreprise nécessaire au déploiement du service web :



- [1], on crée une application d'entreprise [ea-rdvmedecins], au départ sans aucun module.
- en [2], on ajoute le module web [serveur-webservice-ejb-dao-jpa-hibernate] précédent
- en [3], le résultat.

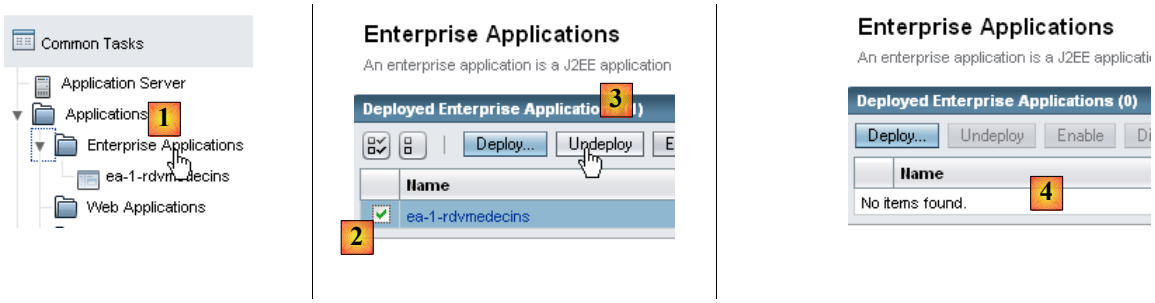
Telle quelle, l'application d'entreprise [ea-rdvmedecins] ne peut pas être déployée sur le serveur Glassfish à partir de Netbeans. On obtient une erreur. Il faut alors déployer à la main l'archive **ear** de l'application [ea-rdvmedecins] :



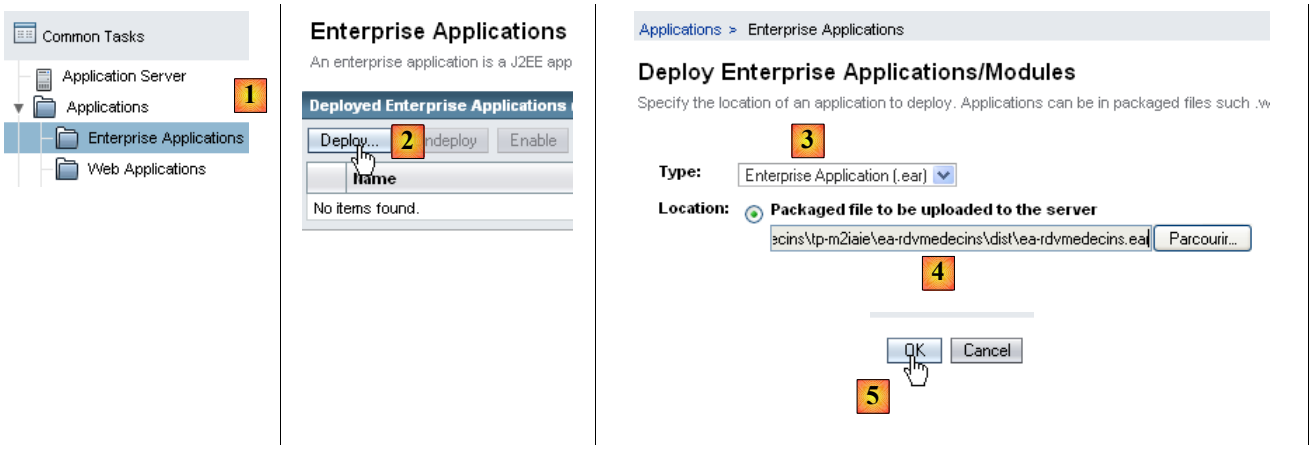
- l'archive [ea-rdvmedecins.ear] est trouvée dans le dossier [dist] [2] de l'onglet [Files] de Netbeans.
- dans cette archive [3], on trouve les deux éléments de l'application d'entreprise :
  - l'archive de l'Ejb [serveur-ejb-dao-jpa-hibernate]. Cette archive est présente parce qu'elle faisait partie des bibliothèques référencées par le service web.
  - l'archive du service web [serveur-webservice- ejb-dao-jpa-hibernate].
- l'archive [ea-rdvmedecins.ear] est construite par un simple *Build* [4] de l'application d'entreprise.
- en [5], l'opération de déploiement qui échoue.

Pour déployer l'archive [ea-rdvmedecins.ear] de l'application d'entreprise, nous procédons comme il a été montré lors du déploiement de l'archive de l'Ejb [serveur-ejb-dao-jpa-hibernate.jar] au paragraphe 4.2, page 8. Nous utilisons de nouveau le client web d'administration du serveur Glassfish. Nous ne répétons pas des étapes déjà décrites.

Tout d'abord, on commencera par "décharger" l'application d'entreprise déployée au paragraphe 4.10.1, page 39 :

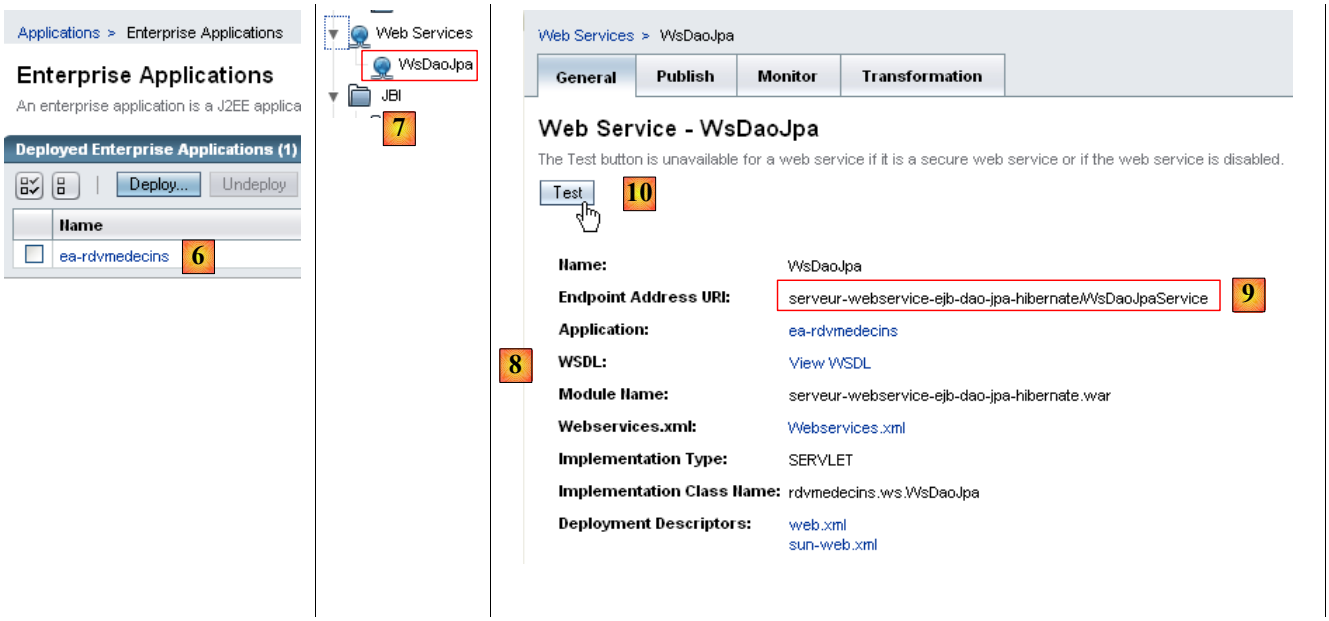


- [1] : sélectionnez la branche [Enterprise Applications] du serveur Glassfish
- en [2] sélectionner l'application d'entreprise à décharger puis en [3] la décharger
- en [4] l'application d'entreprise a été déchargée



- en [1], choisissez la branche [Enterprise Applications] du serveur Glassfish
- en [2], déployez une nouvelle application d'entreprise
- en [3], sélectionnez le type [Enterprise Application]

- en [4], désignez le fichier .ear du projet Netbeans [ea-rdvmedecins]
- en [5], déployez cette archive



- en [6], l'application a été déployée
- en [7], le service web [WsDaoJpa] apparaît dans la branche [Web Services] du serveur Glassfish. On le sélectionne.
- en [8], on a diverses informations sur le service web. La plus intéressante pour un client est l'information [9] : l'uri du service web.
- en [10], on peut tester le service web

- en [11], l'uri du service web à laquelle on a ajouté le paramètre **?tester**. Cette uri présente une page de test. Toutes les méthodes (**@WebMethod**) exposées par le service web sont affichées et peuvent être testées. Ici, on teste la méthode [13] qui demande la liste des clients.

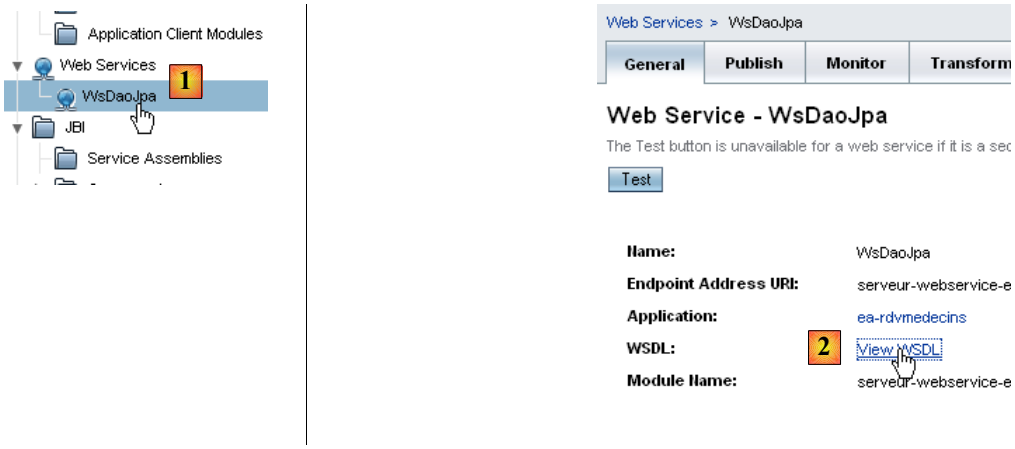
SOAP Response

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

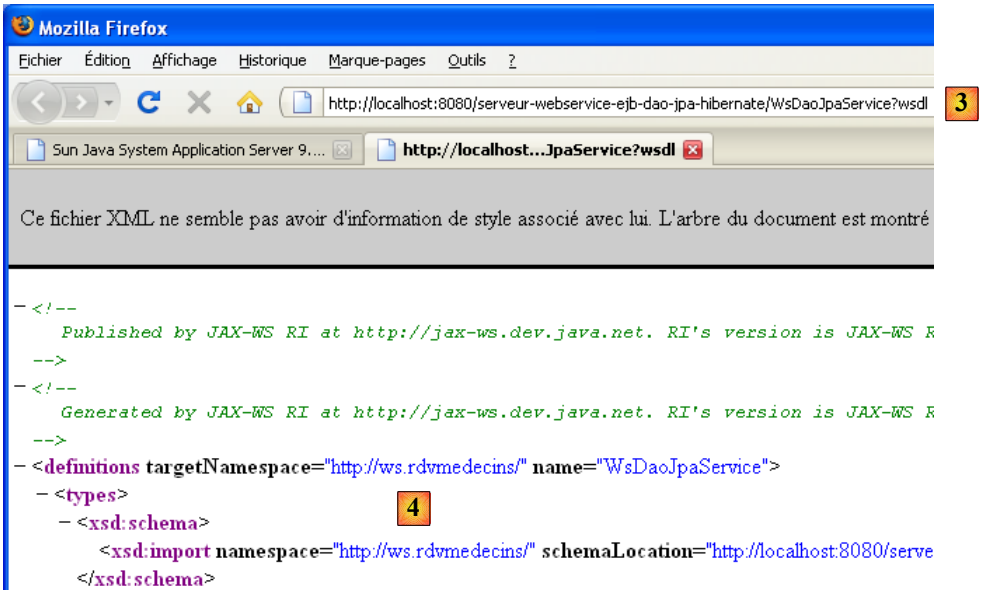
14

- en [14], nous ne présentons qu'une vue partielle de la page de réponse. Mais on peut voir que la méthode **getAllClients** a bien renvoyé la liste des clients. La copie d'écran nous montre qu'elle envoie sa réponse dans un format XML.

Un service web est entièrement décrit par un fichier XML appelé fichier WSDL :



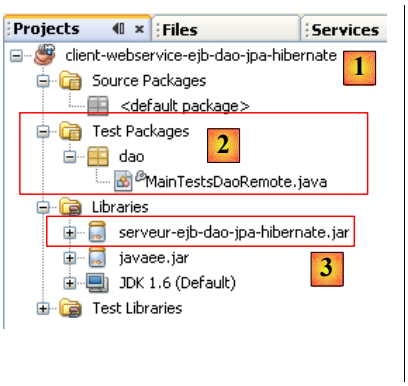
- en [1] dans l'outil web d'administration du serveur Glassfish, sélectionnez le service web [WsDaoJpa]
- en [2], suivez le lien [View WSDL]



- en [3] : l'uri du fichier WSDL. C'est une information importante à connaître. Elle est nécessaire pour configurer les clients de ce service web.
- en [4], la description XML du service web. Nous ne commenterons pas ce contenu complexe.

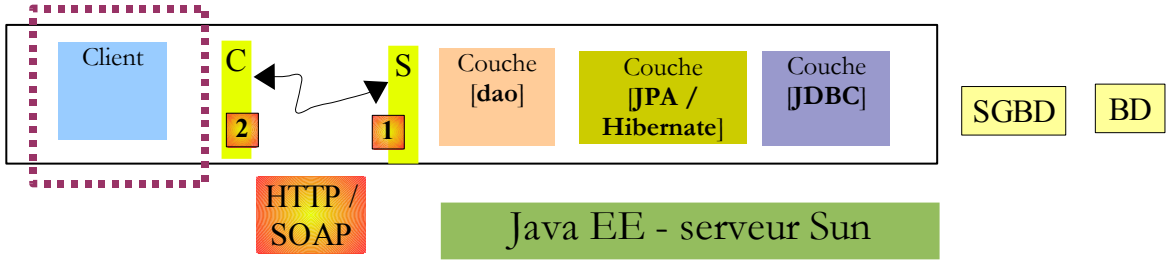
4.10.3 Tests JUnit du service web

Nous créons un projet Netbeans pour "jouer" les tests déjà joués avec un client Ejb avec, cette fois-ci, un client pour le service web dernièrement déployé. Nous suivons ici une démarche analogue à celle décrite au paragraphe 14.2.1, page 115 de [**ref1**].



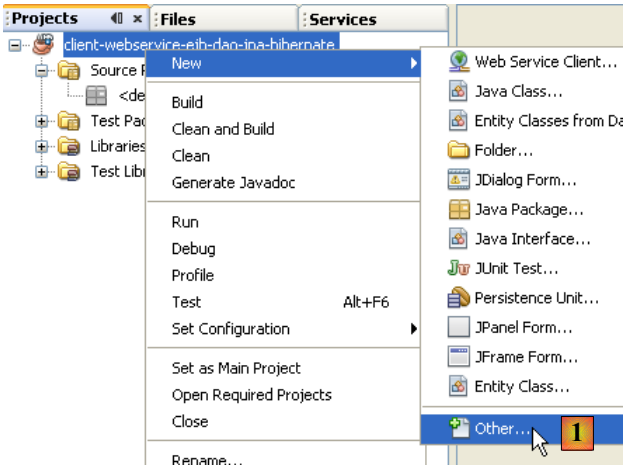
- en [1], un projet Java classique
- en [2], la classe de test
- en [3], le client utilise l'archive de l'Ejb pour avoir accès aux définitions de l'interface de la couche [dao] et des entités Jpa. On rappelle que cette archive est dans le sous-dossier [dist] du dossier du module Ejb.

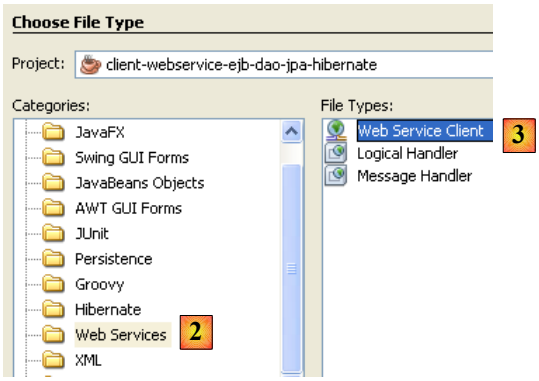
Pour accéder au service web distant, il est nécessaire de générer des classes proxy :



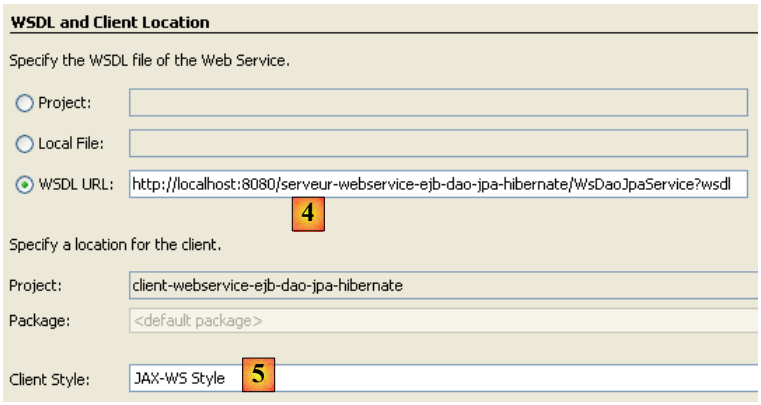
Dans le schéma ci-dessus, la couche [2] [C=Client] communique avec la couche [1] [S=Serveur]. Pour dialoguer avec la couche [S], le client [C] est amené à créer une connexion réseau avec la couche [S] et à dialoguer avec elle selon un protocole précis. Les connexions réseau sont des connexions TCP et le protocole de transport est HTTP. La couche [S] qui représente le service web est implémentée par une servlet Java exécutée par le serveur Glassfish. Nous n'avons pas écrit cette servlet. Sa génération est automatisée par Glassfish à partir des annotations **@Webservice** et **@WebMethod** de la classe [WsDaoJpa] que nous avons écrite. De même, nous allons automatiser la génération de la couche [C] du client. On appelle parfois la couche [C], une couche **proxy** du service web distant, le terme *proxy* désignant un élément intermédiaire dans une chaîne logicielle. Ici, le proxy C est l'intermédiaire entre le client que nous allons écrire et le service web que nous avons déployé.

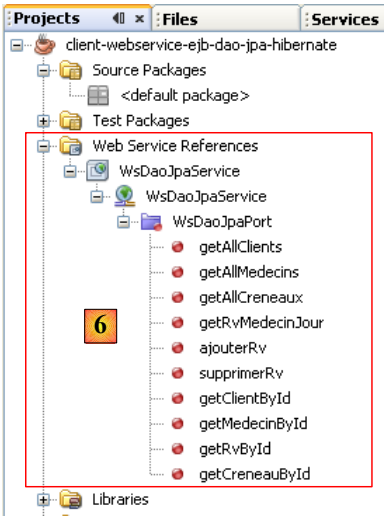
Avec Netbeans 6.5, le proxy C peut être généré de la façon suivante (pour la suite, il faut que le service web soit actif sur le serveur Glassfish) :



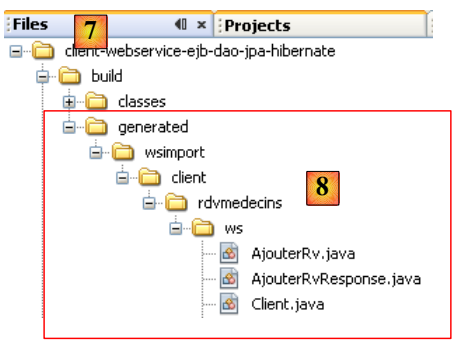


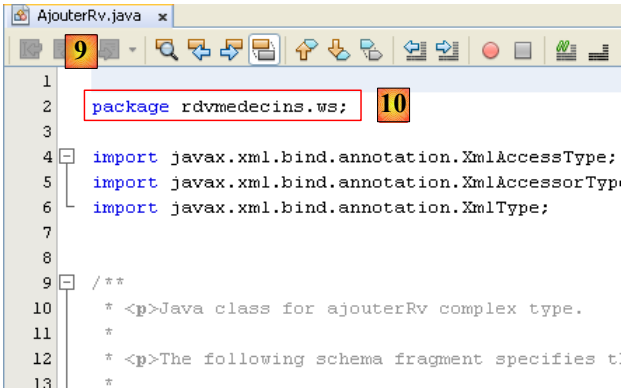
- en [1], ajouter un nouvel élément au projet Java
- en [2], sélectionner la branche [Web services]
- en [3], sélectionner [Web Service Client]





- en [4], fournir l'uri du fichier WSDL du service web. Cette uri a été présentée au paragraphe 4.10.2, page 46.
- en [5], laisser la valeur par défaut [JAX-WS]. L'autre valeur possible est [JAX-RPC]
- après avoir validé l'assistant de création du proxy du service web, le projet Netbeans a été enrichi d'une branche [Web Service References] [6]. Cette branche montre les méthodes exposées par le service web distant.

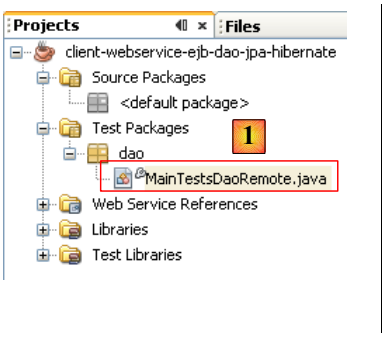




- dans l'onglet [Files] [7], des codes sources Java ont été rajoutés [8]. Ils correspondent au proxy C généré.
- en [9] le code de l'une des classes. On y voit [10] qu'elles ont été placées dans un paquetage [rdvmedecins.ws]. Nous ne commenterons pas le code de ces classes qui est de nouveau assez complexe.

Pour le client Java que nous sommes en train de construire, le proxy C généré sert d'intermédiaire. Pour accéder à la méthode M du service web distant, le client Java appelle la méthode M du proxy C. Le client Java appelle ainsi des méthodes locales (exécutées dans la même Jvm) et de façon transparente pour lui, ces appels locaux sont traduits en appels distants.

Il nous reste à savoir appeler les méthodes M du proxy C. Revenons à notre classe de test JUnit :

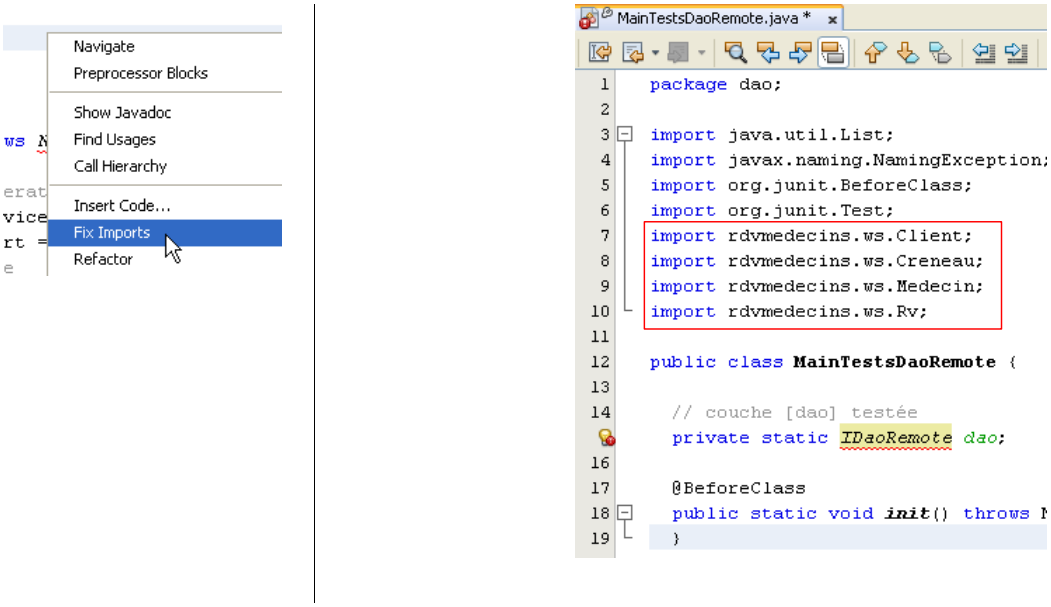


En [1], la classe de test [MainTestsDaoRemote] est celle déjà utilisée lors du test de l'Ejb de la couche [dao] :

```
1.  package dao;
2.  ...
3.  public class MainTestsDaoRemote {
4.
5.    // couche [dao] testée
6.    private static IDaoRemote dao;
7.
8.    @BeforeClass
9.    public static void init() throws NamingException {
10.   }
11.
12.   @Test
13.   public void test1() {
14. ...
15.   }
16. }
```

- ligne [13], le test *test1* est conservé à l'identique.
- ligne [9], le contenu de la méthode [init] a été supprimé.

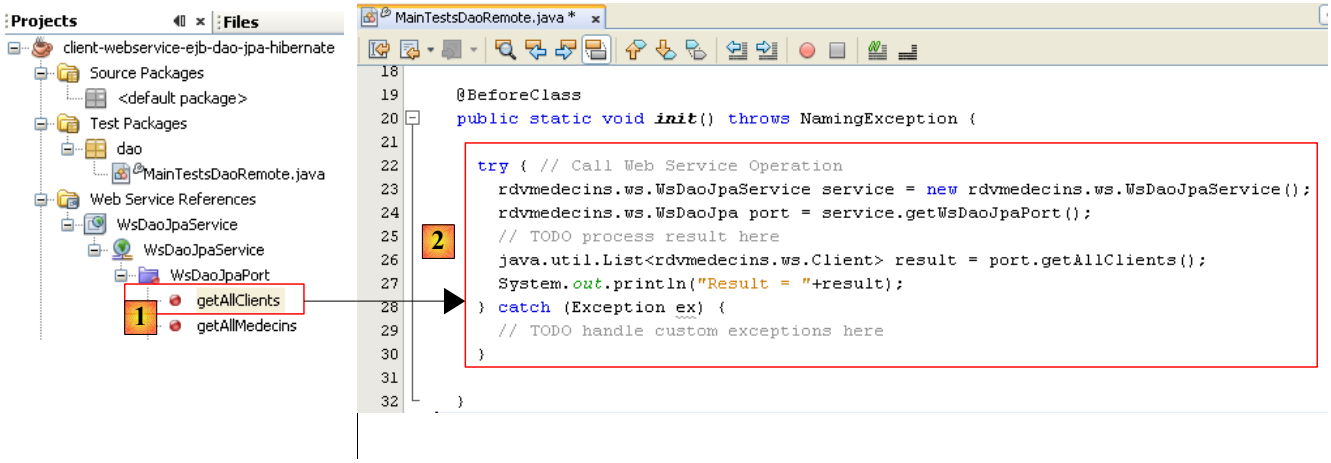
A ce stade, le projet présente des erreurs car la méthode de test [test1] utilise les entités [Client], [Medecin], [Creneau], [Rv] qui ne sont plus dans les mêmes packages qu'auparavant. Elles sont dans le package du proxy C généré. On supprime les instructions *import* concernées et on les régénère par l'opération *Fix Imports*.



Revenons au code de la classe de test [MainTestsDaoRemote] :

```
1.  package dao;
2.  ...
3.
4.  public class MainTestsDaoRemote {
5.
6.    // couche [dao] testée
7.    private static IDaoRemote dao;
8.
9.    @BeforeClass
10.   public static void init() throws NamingException {
11. }
```

La méthode [init] de la ligne 10 doit initialiser la référence de la couche [dao] de la ligne 7. Il nous faut savoir comment utiliser le proxy C généré, dans notre code. Netbeans nous aide dans cette démarche.



- sélectionner en [1] la méthode [getAllClients] du service web puis, avec la souris, tirer cette méthode pour aller la déposer au sein de la méthode [init] de la classe de test.

On obtient le résultat [2]. Ce squelette de code nous montre comment utiliser le proxy C généré :

```
1.      try { // Call Web Service Operation
2.        rdvmedecins.ws.WsDaoJpaService service = new rdvmedecins.ws.WsDaoJpaService();
3.        rdvmedecins.ws.WsDaoJpa port = service.getWsDaoJpaPort();
4.        // TODO process result here
5.        java.util.List<rdvmedecins.ws.Client> result = port.getAllClients();
```

```
6.        System.out.println("Result = "+result);
7.      } catch (Exception ex) {
8.        // TODO handle custom exceptions here
9.  }
```

- la ligne [5] nous montre que la méthode [getAllClients] est une méthode de l'objet de type [WsDaoJpa] défini ligne 3. Le type [WsDaoJpa] est une interface présentant les mêmes méthodes que celles exposées par le service web distant.
- ligne [3], l'objet [WsDaoJpa port] est obtenu à partir d'un autre objet de type [WsDaoJpaService] défini ligne 2. Le type [WsDaoJpaService] représente le proxy C généré localement.
- l'accès au service web distant peut échouer, aussi l'ensemble du code est-il entouré d'un *try / catch*.
- les objets du proxy C sont dans le package [rdvmedecins.ws]

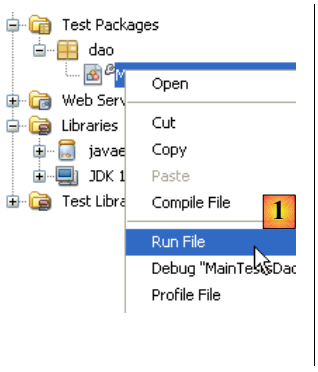
Une fois ce code compris, on voit que la référence locale du service web distant peut être obtenue par le code :

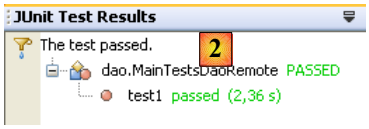
```
WsDaoJpa dao=new WsDaoJpaService().getWsDaoJpaPort();
```

Le code de la classe de test JUnit devient alors le suivant :

```
1.  package dao;
2.
3.  import rdvmedecins.ws.Client;
4.  import rdvmedecins.ws.Creneau;
5.  import rdvmedecins.ws.Medecin;
6.  import rdvmedecins.ws.Rv;
7.  import rdvmedecins.ws.WsDaoJpa;
8.  import rdvmedecins.ws.WsDaoJpaService;
9.  ...
10.
11. public class MainTestsDaoRemote {
12.
13.   // couche [dao] testée
14.   private static WsDaoJpa dao;
15.
16.   @BeforeClass
17.   public static void init(){
18.     dao=new WsDaoJpaService().getWsDaoJpaPort();
19.   }
20.
21.   @Test
22.   public void test1() {
23. ...
24.   }
25.
26.   // méthode utilitaire - affiche les éléments d'une collection
27.   private static void display(String message, List elements) {
28.  ...
29.   }
30. }
```

Nous sommes désormais prêts pour les tests :





En [1], le test JUnit est exécuté. En [2], il est réussi. Si on regarde les affichages sur la console Netbeans, on trouve des lignes comme les suivantes :

```
Liste des clients :
rdvmedecins.ws.Client@1982fc1
```

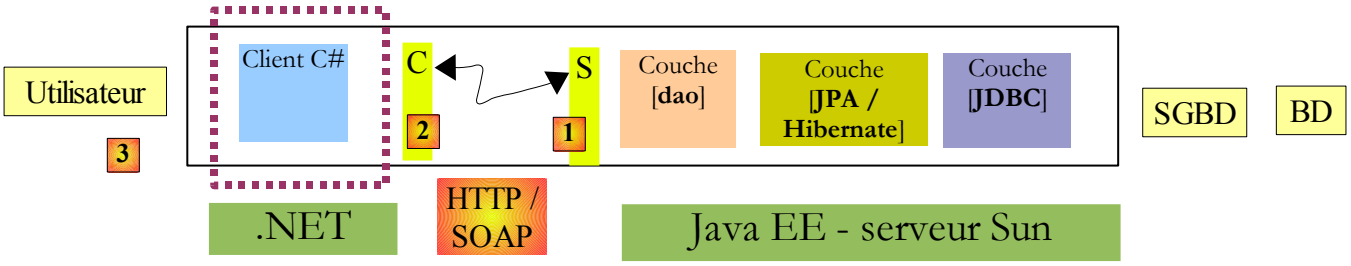
```
rdvmedecins.ws.Client@676437
rdvmedecins.ws.Client@1e4853f
rdvmedecins.ws.Client@1e808ca
```

Côté serveur, l'entité [Client] a une méthode *toString* qui affiche les différents champs d'un objet de type [Client]. Lors de la génération automatique du proxy C, les entités sont créées dans le proxy C mais avec seulement les champs privés accompagnés de leurs méthodes *get* / *set*. Ainsi la méthode *toString* n'a pas été générée dans l'entité [Client] du proxy C. Ce qui explique l'affichage précédent. Ceci n'enlève rien au test JUnit : il a été réussi. On considèrera désormais qu'on a un service web opérationnel.

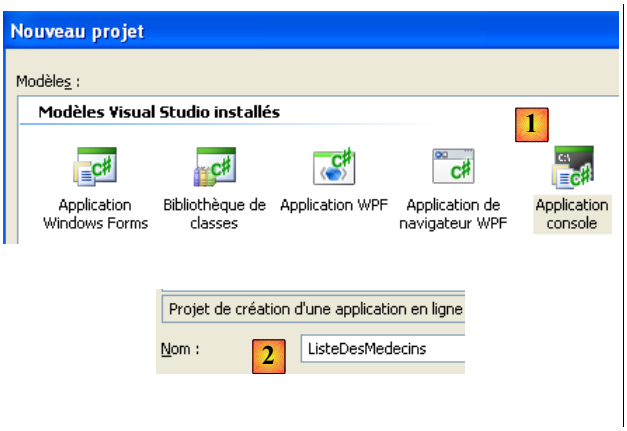
5 Clients .NET du service Java EE des rendez-vous

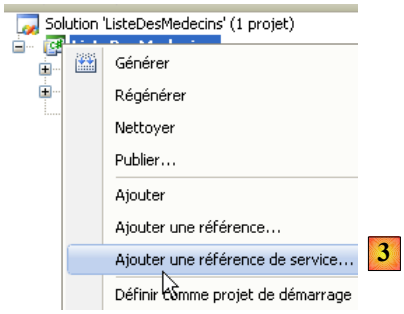
5.1 Un client C# 2008

Nous supposons désormais que le service web précédent est disponible et actif. Un service web est utilisable par des clients qui peuvent être écrits en différents langages. Nous nous proposons ici d'écrire un client console C# pour afficher la liste des médecins.

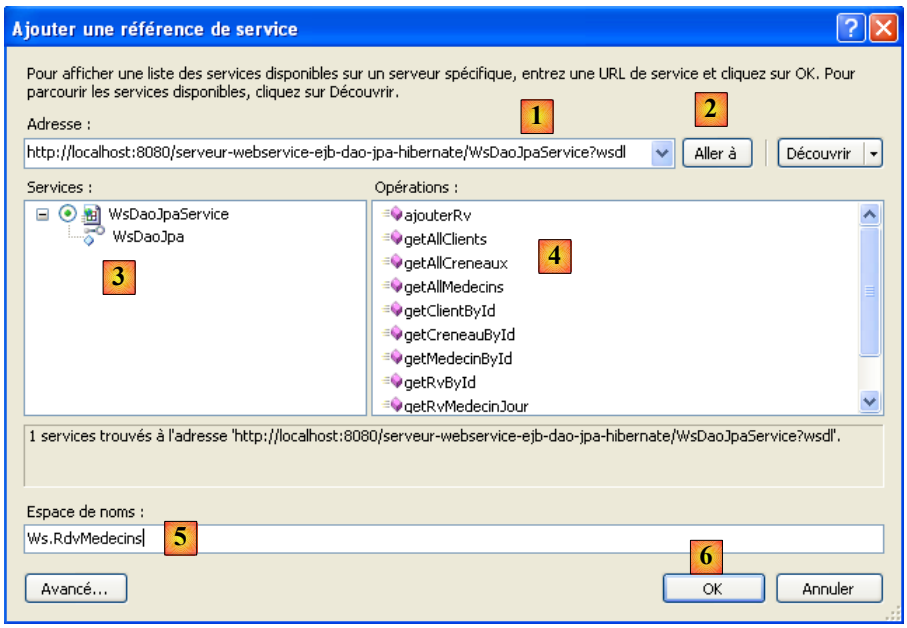


Commençons par créer le projet C# :

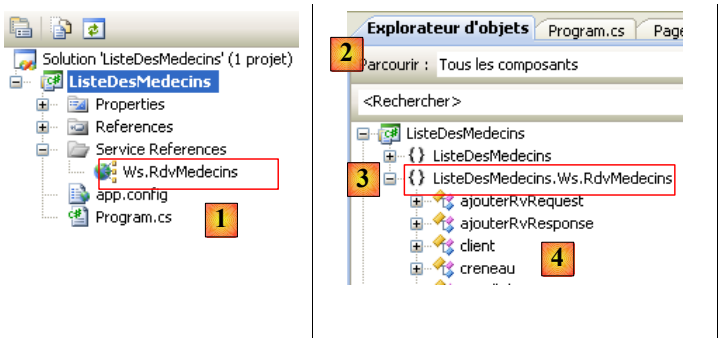




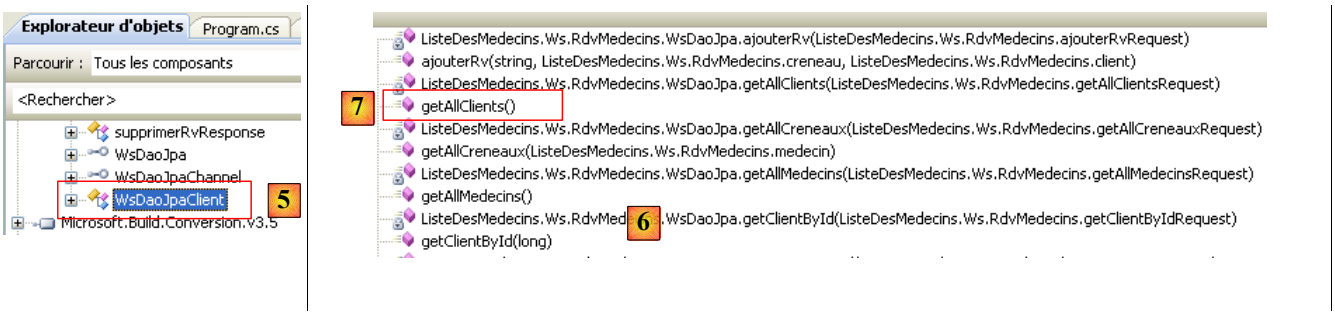
On crée une application console [1] appelée [ListeDesMedecins] [2]. En [3], on crée une référence sur un service web. Cet outil est similaire à celui utilisé dans Netbeans : il crée un proxy local du service web distant.



- en [1], on indique l'uri du fichier WSDL du service web (cf paragraphe 4.10.2, page 46).
- en [2], on demande à l'assistant de découvrir les services web exposés à cette uri
- en [3], le service web trouvé et en [4] les méthodes qu'il expose.
- en [5], on précise dans quel espace de noms, les classes du proxy C doivent être générées
- en [6], on génère le proxy C.

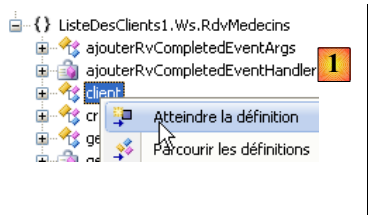


- en [1], la référence web que nous venons de créer. Double-cliquons dessus.
- en [2], l'explorateur d'objets ouvert par le double-clic.
- en [3], on sélectionne l'espace de noms [ListeDesMedecins.Ws.RdvMedecins] qui est l'espace de noms du proxy C généré. Le 1er terme [ListeDesMedecins] est l'espace de noms par défaut de l'application C# créée (nous avons appelée celle-ci *ListeDesMedecins*). Le second terme [Ws.Rdvmedecins] est l'espace de noms que nous avons donné au proxy C dans l'assistant. Au final, les objets du proxy C sont l'espace de noms [ListeDesMedecins.Ws.RdvMedecins].
- en [4], les objets du proxy C généré



- en [5], [WsDaoJpaClient] est la classe implémentant localement les méthodes du service web distant.
- en [6], les méthodes de la classe [WsDaoJpaClient]. On y retrouve les méthodes du service web distant commepar exemple la méthode *getAllClients* en [7].

Examinons le code des entités générées :



En [1] ci-dessus, on demande à voir le code de la classe [client] :

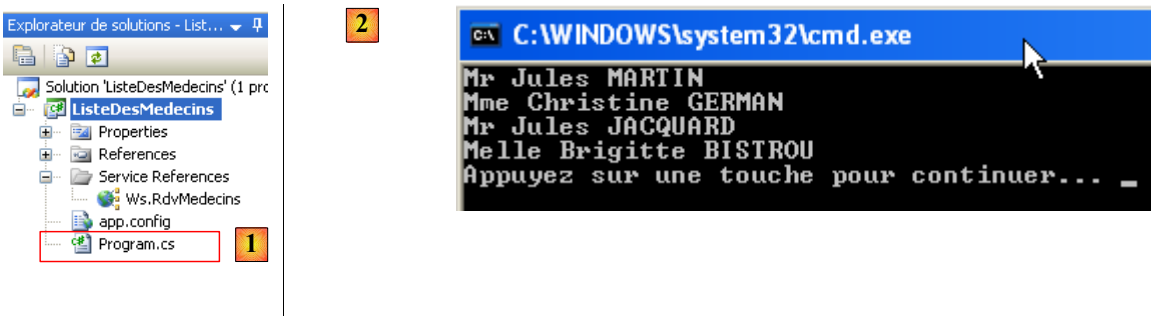
```
1.  ...
2.      public partial class client : personne {
3.      }
4.
5.  ....
6.      public partial class personne : object, System.ComponentModel.INotifyPropertyChanged {
7.      ...
8.          public long id {
9.          ...
10.         }
11.
12. ...
13. }
```

Ci-dessus, nous n'avons gardé que le code nécessaire à notre étude.

- ligne 2 : la classe [client] dérive de la classe [personne] comme dans le service web.
- ligne 8 : une propriété publique de la classe [personne]

Ce qu'on remarque, c'est que le code généré ne respecte pas les normes de codage habituelles de C#. Les noms des classes et des propriétés publiques devraient commencer par une majuscule.

Revenons à notre application C# :



[Program.cs] [1] est la classe de test. Son code est le suivant :

```
1.  using System;
2.  using ListeDesMedecins.Ws.RdvMedecins;
3.  using Client = ListeDesMedecins.Ws.RdvMedecins.client;
4.
5.  namespace ListeDesMedecins {
6.     class Program {
7.        static void Main(string[] args) {
8.           try {
9.              // instanciation couche [dao] du client
10.             WsDaoJpaClient dao = new WsDaoJpaClient();
11.             // liste des médecins
12.             foreach (Client client in dao.getAllClients()) {
13.                Console.WriteLine(String.Format("{0} {1} {2}", client.titre, client.prenom, client.nom));
14.             }
```

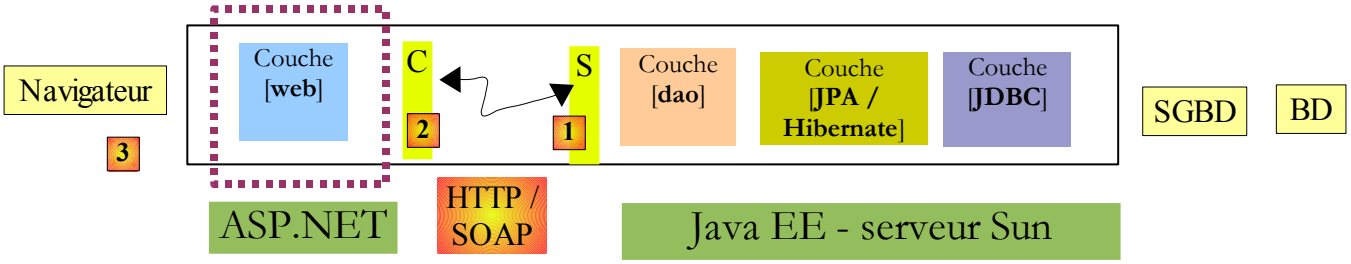
```
15.             } catch (Exception e) {
16.                 Console.WriteLine(e);
17.             }
18.         }
19.     }
20. }
```

Ligne 12, la classe [Client] du proxy C est utilisée alors que nous venons de voir qu'elle s'appelait en réalité [client]. C'est la déclaration de la ligne 3 qui nous permet d'utiliser [Client] au lieu de [client]. Cela nous permet de respecter la norme de codage des noms de classe. Toujours ligne 12, la méthode [getAllClients] est utilisée parce c'est ainsi qu'elle s'appelle dans le proxy C. La norme de codage C# voudrait qu'elle s'appelle [GetAllClients].

L'exécution du programme [1] précédent donne les résultats montrés en [2].

5.2 Un premier client ASP.NET 2008

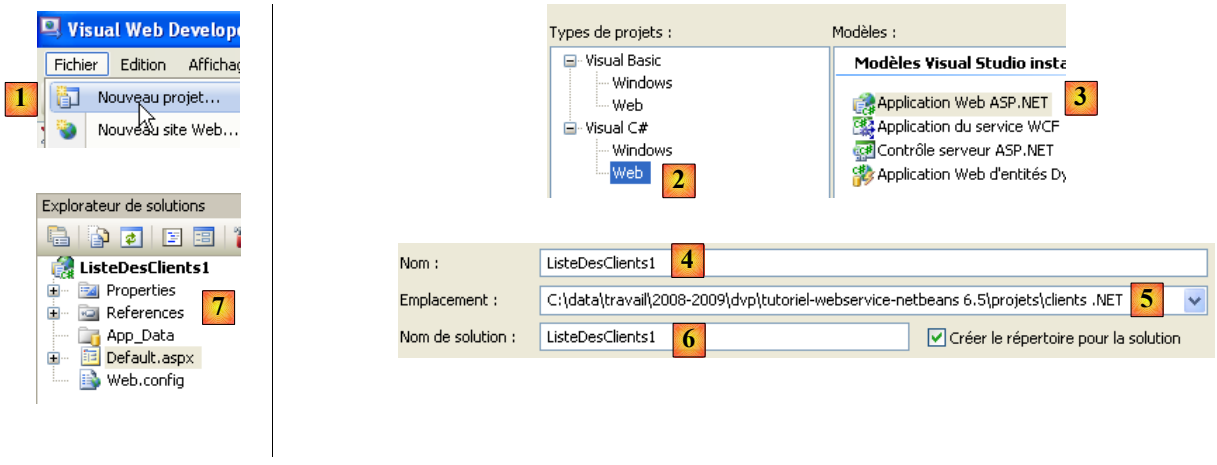
Nous écrivons maintenant un premier client ASP.NET / C# pour afficher la liste des clients.



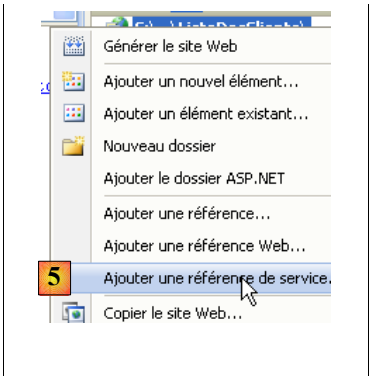
Dans cette architecture, il y a deux serveurs web :

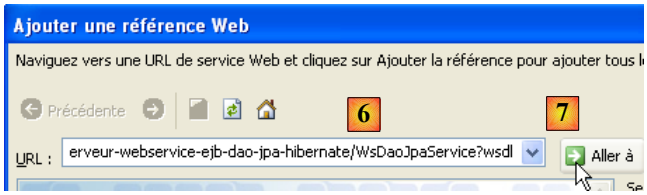
- celui qui exécute le service web distant
- celui qui exécute le client ASP.NET du service web distant

Nous allons créer le projet web du client ASP.NET avec la version Visual Web Express 2008 **SP1**. Le paragraphe suivant montre comment créer le même projet si on n'a pas la version **SP1** de Visual Web Express 2008.

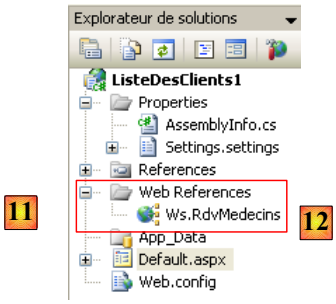


On crée une application *web* [1, 2, 3] appelée [ListeDesClients1] [4,5,6]. Le projet ainsi créé est visible en [7].



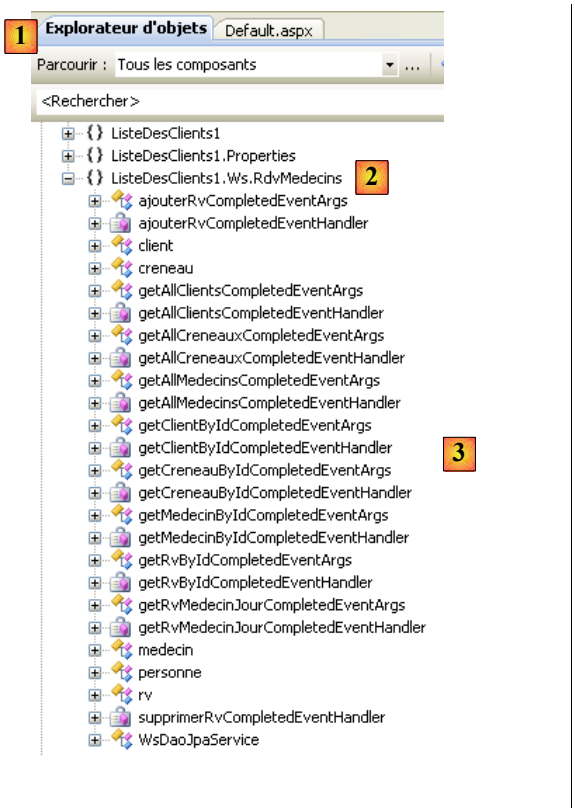


- en [5], par un clic droit sur le nom du projet, on ajoute une référence de service web.
- en [6], on indique l'uri du fichier WSDL du service web (cf paragraphe 4.10.2, page 46).
- en [7], on demande à l'assistant de découvrir les services web exposés à cette uri

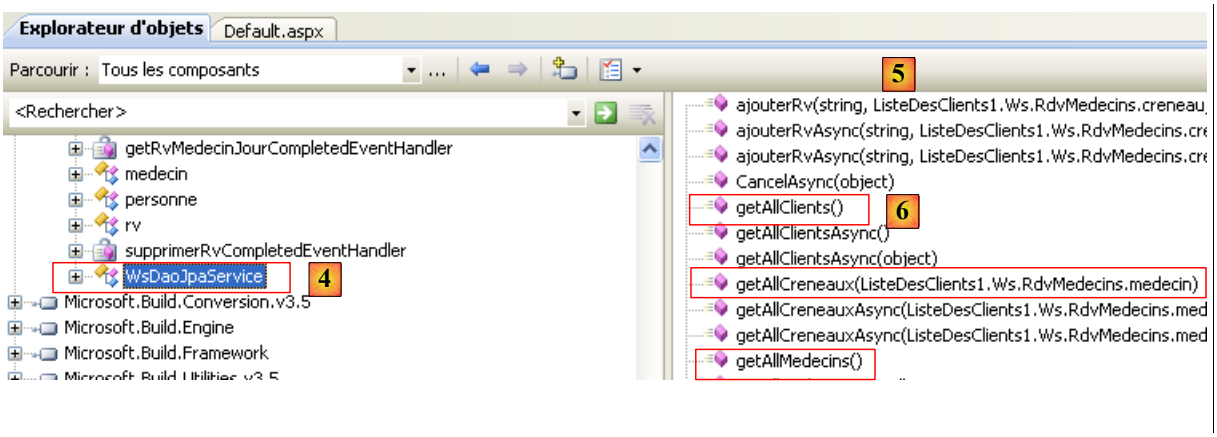


- en [8], le service web trouvé et en [9] les méthodes qu'il expose sont affichées.
- en [10], on précise dans quel espace de noms, les classes du proxy C doivent être générées
- en [11], la référence du service web générée

Un double-clic sur la référence [Ws.Rdvmedecins] [12] affiche les classes et interfaces générées pour le proxy C :

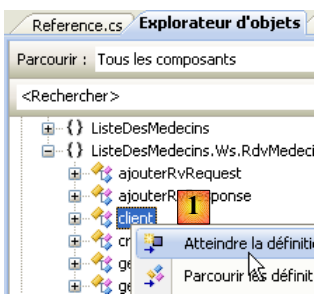


- en [1], l'explorateur d'objets ouvert par le double-clic.
- en [2], on sélectionne l'espace de noms [ListeDesClients1.Ws.RdvMedecins] qui est l'espace de noms du proxy C généré. Le 1er terme [ListeDesClients1] est l'espace de noms par défaut de l'application ASP.NET créée (nous avons appelée celle-ci *ListeDesClients1*). Le second terme [Ws.Rdvmedecins] est l'espace de noms que nous avons donné au proxy C dans l'assistant. Au final, les objets du proxy C sont l'espace de noms [ListeDesClients1.Ws.RdvMedecins].
- en [3], les objets du proxy C généré



- en [4], [WsDaoJpaService] est la classe implémentant localement les méthodes du service web distant.
- en [5], les méthodes de la classe [WsDaoJpaService]. On y retrouve les méthodes du service web distant comme par exemple la méthode *getAllClients* en [6].

Examinons le code des entités générées :



En [1] ci-dessus, on demande à voir le code de la classe [client] :

```
1.  public partial class client : personne {
2.      }
3.
4.  ...
5.      public partial class personne {
6.    ...
7.          public long id {
8.  ...
9.          }
```

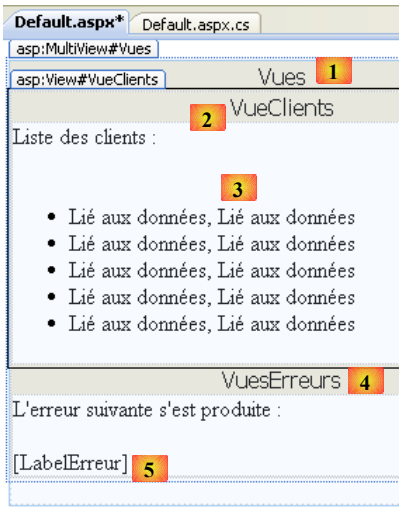
Ci-dessus, nous n'avons gardé que le code nécessaire à notre étude.

- ligne 1 : la classe [client] dérive de la classe [personne] comme dans le service web.
- ligne 5 : la classe [personne]
- ligne 7 : une propriété publique de la classe [personne]

On retrouve un code analogue à celui déjà commenté pour le client C# à la page 55.

- l'espace de noms du proxy C généré est celui défini dans son assistant de création préfixé par l'espace de noms du projet web lui-même : **ListeDesClients1.Ws.RdvMedecins**
- la classe proxy qui implémente localement les méthodes du service web distant s'appelle **WsDaoJpaService**.
- les noms des classes et des propriétés commencent par une minuscule

Nous pouvons écrire une page [Default.aspx] affichant la liste des clients. La page est la suivante :



N°	Type	Nom	Rôle
1	MultiView	Vues	Conteneur de vues

N°	Type	Nom	Rôle
2	View	VueClients	la vue qui contient la liste des clients
3	Repeater	RepeaterClients	affiche la liste des clients sous la forme d'une liste à puces
4	View	VueErreurs	la vue qui affiche une éventuelle erreur
5	Label	LabelErreur	le texte de l'erreur

Le code de présentation de cette page est comme suit :

```
1.  <%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Default.aspx.cs"
    Inherits="ListeDesClients1._Default" %>
2.
3.  <%@ Import Namespace="ListeDesClients1.Ws.RdvMedecins" %>
4.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
5.  <html xmlns="http://www.w3.org/1999/xhtml">
6.  <head id="Head1" runat="server">
7.    <title>Liste des clients</title>
8.  </head>
9.  <body>
10.   <form id="form1" runat="server">
11.   <div>
12.     <asp:MultiView ID="Vues" runat="server">
13.       <asp:View ID="VueClients" runat="server">
14.         Liste des clients :<br />
15.         <br />
16.         <ul>
17.           <asp:Repeater ID="RepeaterClients" runat="server">
18.             <ItemTemplate>
19.               <li>
20.                 <%#(Container.DataItem as client).nom%>,
21.                 <%#(Container.DataItem as client).prenom%>
22.               </li>
23.             </ItemTemplate>
24.           </asp:Repeater>
25.         </ul>
26.       </asp:View>
27.       <asp:View ID="VuesErreurs" runat="server">
28.         L'erreur suivante s'est produite :<br />
29.         <br />
30.         <asp:Label ID="LabelErreur" runat="server"></asp:Label></asp:View>
31.     </asp:MultiView><br />
32.   </div>
33.   </form>
34. </body>
35. </html>
```

On notera, ligne 3, la nécessité d'importer l'espace de noms [ListeDesClients1.Ws.RdvMedecins] du proxy C du service web afin que la classe [client] des lignes 20 et 21 soit accessible.

Le code de contrôle [Default.aspx.cs] associé à la page de présentation [Default.aspx] est le suivant :

```
1.  using System;
2.  using ListeDesClients1.Ws.RdvMedecins;
3.
4.  namespace ListeDesClients1
5.  {
6.    public partial class _Default : System.Web.UI.Page
7.    {
8.      protected void Page_Load(object sender, EventArgs e)
9.      {
10.       try
11.       {
12.         // instanciation proxy local de la couche [dao] distante
13.         WsDaoJpaService dao = new WsDaoJpaService();
14.         // vue client
15.         Vues.ActiveViewIndex = 0;
16.         // initialisation vue client
17.         RepeaterClients.DataSource = dao.getAllClients();
18.         RepeaterClients.DataBind();
19.       }
20.       catch (Exception ex)
21.       {
22.         // vue erreurs
23.         Vues.ActiveViewIndex = 1;
```

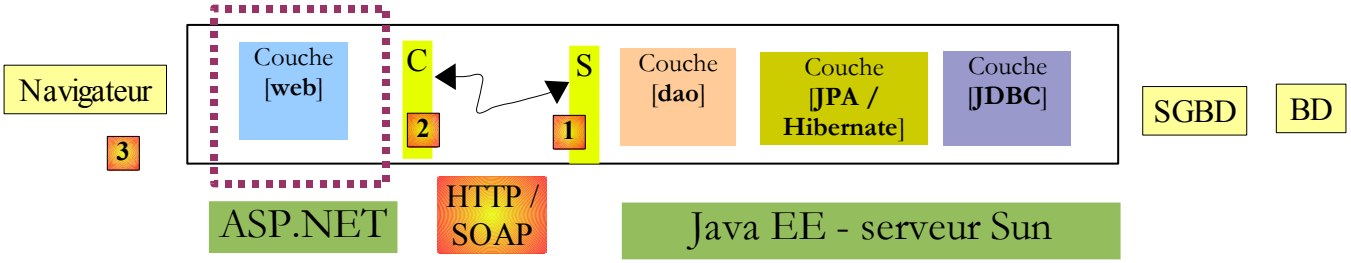
```
24.         //initialisation vue erreurs
25.         LabelErreur.Text = ex.ToString();
26.       }
27.     }
28.   }
29. }
```

- ligne 8 : la méthode *Page_Load* est exécutée au chargement initial de la page. Ce chargement a lieu à chaque requête d'un client demandant la page.
- ligne 13 : on instancie le proxy local C. Ce proxy C implémente l'interface du service web distant. L'application va dialoguer avec ce proxy C.
- ligne 15 : si pas d'exception lors de l'instanciation du proxy local C, c'est la vue nommée "VueClients" qui doit être affichée, c.a.d. la vue n° 0 dans le MultiView "Vues" de la page.
- ligne 17 : la source de données du *répéteur* est la liste des clients fournie par la méthode *getAllClients* du proxy C.
- ligne 23 : en cas d'exception lors de l'instanciation du proxy local C, c'est la vue nommée "VueErreurs" qui doit être affichée, c.a.d. la vue n° 1 dans le MultiView "Vues" de la page.
- ligne 25 : l'exception est affichée dans le *label* "LabelErreur".

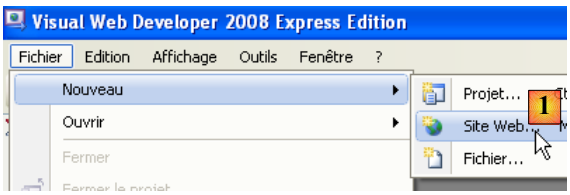
L'exécution du projet web donne le résultat suivant :

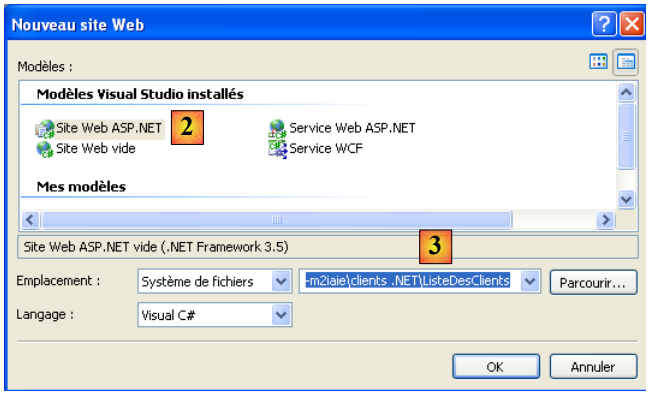
5.3 Un second client ASP.NET 2008

Nous écrivons maintenant un second client ASP.NET / C# avec cette fois-ci la version Visual Web Developer Express 2008 qui a précédé la version **SP1**.

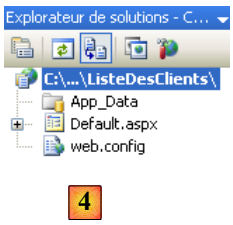


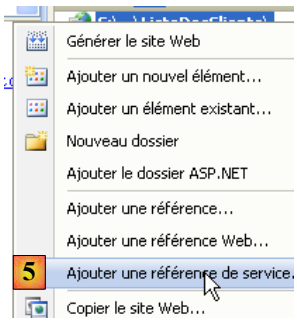
Créons le projet web du client .NET :

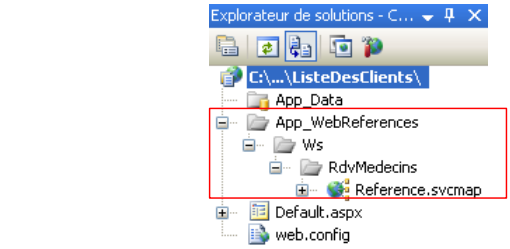




On crée un site *web* [1, 2] appelée [ListeDesClients] [3].







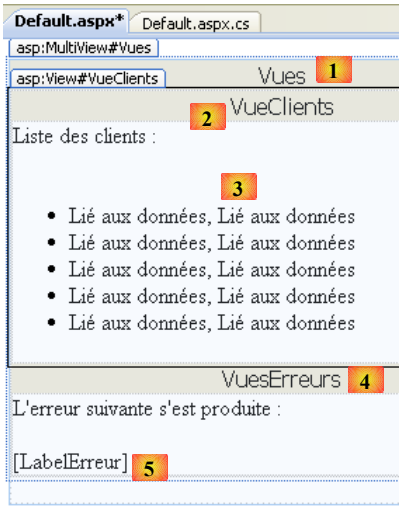
- en [4], le projet web
- en [5], par un clic droit sur le nom du projet, on ajoute une référence de service comme il a été fait précédemment.
- en [6], le projet une fois la référence au service web distant ajoutée

Contrairement au projet web SP1 étudié précédemment, on n'a pas accès aux objets du client du service web via l'explorateur d'objets.

Ce qu'il faut savoir :

- l'espace de noms du proxy C généré est celui défini dans son assistant de création : **Ws.RdvMedecins**
- la classe proxy qui implémente localement les méthodes du service web distant s'appelle **WsDaoJpaClient**.
- les noms des classes et des propriétés commencent par une minuscule

Nous pouvons écrire une page [Default.aspx] affichant la liste des clients. La page est la suivante :



N°	Type	Nom	Rôle
1	MultiView	Vues	Conteneur de vues
2	View	VueClients	la vue qui contient la liste des clients
3	Repeater	RepeaterClients	affiche la liste des clients sous la forme d'une liste à puces
4	View	VueErreurs	la vue qui affiche une éventuelle erreur
5	Label	LabelErreur	le texte de l'erreur

Le code de présentation de cette page est comme suit :

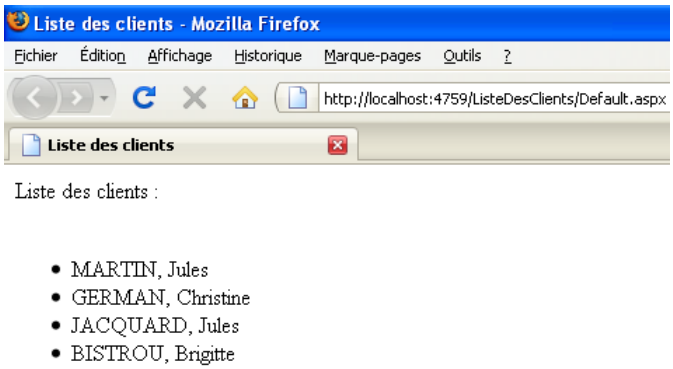
```
1.  <%@ Page Language="C#" AutoEventWireup="true"  CodeFile="Default.aspx.cs" Inherits="_Default" %>
2.  <%@ Import Namespace="Ws.RdvMedecins" %>
3.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4.  <html xmlns="http://www.w3.org/1999/xhtml">
5.  <head id="Head1" runat="server">
6.     <title>Liste des clients</title>
7.  </head>
8.  <body>
9.     <form id="form1" runat="server">
10.       <div>
11.          <asp:MultiView ID="Vues" runat="server">
12.             <asp:View ID="VueClients" runat="server">
13.                Liste des clients :<br />
14.                <br />
15.                <ul>
16.                   <asp:Repeater ID="RepeaterClients" runat="server">
17.                      <ItemTemplate>
18.                         <li>
19.                            <%#(Container.DataItem as client).nom%>, <%#(Container.DataItem as
    client).prenom%>
20.                         </li>
21.                      </ItemTemplate>
22.                   </asp:Repeater>
23.                </ul>
24.             </asp:View>
25.             <asp:View ID="VuesErreurs" runat="server">
26.                L'erreur suivante s'est produite :<br />
27.                <br />
28.                <asp:Label ID="LabelErreur" runat="server"></asp:Label></asp:View>
29.          </asp:MultiView><br />
30.       </div>
31.    </form>
32. </body>
33. </html>
```

On notera, ligne 2, la nécessité d'importer l'espace de noms [Ws.RdvMedecins] du proxy C du service web afin que la classe [client] de la ligne 19 soit accessible.

Le code de contrôle [Default.aspx.cs] associé à la page de présentation [Default.aspx] est le suivant :

```
1.  using System;
2.  using Ws.RdvMedecins;
3.
4.  public partial class _Default : System.Web.UI.Page
5.  {
6.    protected void Page_Load(object sender, EventArgs e)
7.    {
8.      try
9.      {
10.       // instanciation proxy local de la couche [dao] distante
11.       WsDaoJpaClient dao = new WsDaoJpaClient();
12.       // vue client
13.       Vues.ActiveViewIndex = 0;
14.       // initialisation vue client
15.       RepeaterClients.DataSource = dao.getAllClients();
16.       RepeaterClients.DataBind();
17.     }
18.     catch (Exception ex)
19.     {
20.       // vue erreurs
21.       Vues.ActiveViewIndex = 1;
22.       //initialisation vue erreurs
23.       LabelErreur.Text = ex.ToString();
24.     }
25.   }
26. }
```

Ce code est analogue à celui de la version précédente. L'exécution du projet web donne le résultat suivant :



6 Clients Flex du service Java EE des rendez-vous

Nous présentons maintenant deux clients **Flex** du service web JEE des rendez-vous. L'IDE utilisé est Flex Builder 3. Une version de démonstration de ce produit est téléchargeable à l'Url [https://www.adobe.com/cfusion/tdrc/index.cfm?loc=fr_fr&product=flex]. Flex Builder 3 est un IDE Eclipse. Par ailleurs, pour exécuter le client Flex, nous utilisons un serveur web Apache de l'outil **Wamp** [http://www.wampserver.com/]. N'importe quel serveur **Apache** fait l'affaire. Le navigateur qui affiche le client Flex doit disposer du plugin **Flash Player** version 9 minimale.

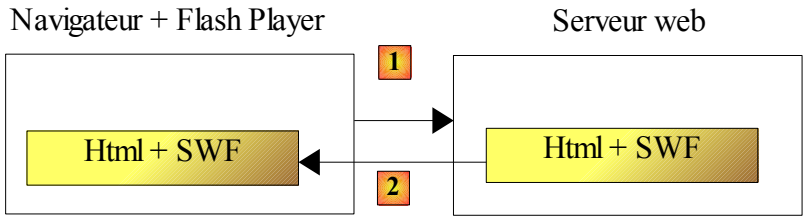
Les applications Flex ont la particularité de s'exécuter au sein du plugin Flash Player du navigateur. En cela elles se rapprochent des applications Ajax qui embarquent dans les pages envoyées au navigateur des scripts JavaScript qui sont ensuite exécutés au sein du navigateur. Une application Flex n'est pas une application web au sens où on l'entend habituellement : c'est une application cliente de services délivrés par des serveurs web. En cela, elle est analogue à une application de bureau qui serait cliente de ces mêmes services. Elle diffère cependant en un point : elle est téléchargée initialement depuis un serveur web au sein d'un navigateur disposant du plugin Flash Player capable de l'exécuter.

Comme une application de bureau, une application Flex est composée principalement de deux éléments :

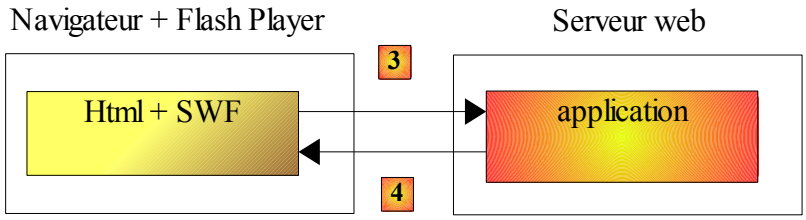
- une partie **présentation** : les vues affichées dans le navigateur. Ces vues ont la richesse des fenêtres des applications de bureau. Une vue est décrite à l'aide d'un langage à balises appelé **MXML**.

- une partie **code** qui gère principalement les événements provoqués par les actions de l'utilisateur sur la vue. Ce code peut être écrit également en **MXML** ou avec un langage orienté objet appelé **ActionScript**. Il faut distinguer deux types d'événements :
  - l'événement qui nécessite d'avoir un échange avec le serveur web : remplissage d'une liste par des données fournies par une application web, envoi des données d'un formulaire au serveur, ... Flex fournit un certain nombre de méthodes pour communiquer avec le serveur de façon transparente pour le développeur. Ces méthodes sont par défaut **asynchrones** : l'utilisateur peut continuer à interagir avec la vue pendant la requête au serveur.
  - l'événement qui modifie la vue affichée sans échange de données avec le serveur, par exemple tirer un élément d'un arbre pour le déposer dans une liste. Ce type d'événement est entièrement traité localement au sein du navigateur.

Une application Flex est souvent exécutée de la façon suivante :



- en [1], une page Html est demandée
- en [2], elle est envoyée. Elle embarque avec elle un fichier binaire **SWF** (**S**hock**W**ave **F**lash)contenant l'intégralité de l'application Flex : toutes les vues et le code de gestion des événements de celles-ci. Ce fichier sera exécuté par le plugin FlashPlayer du navigateur.



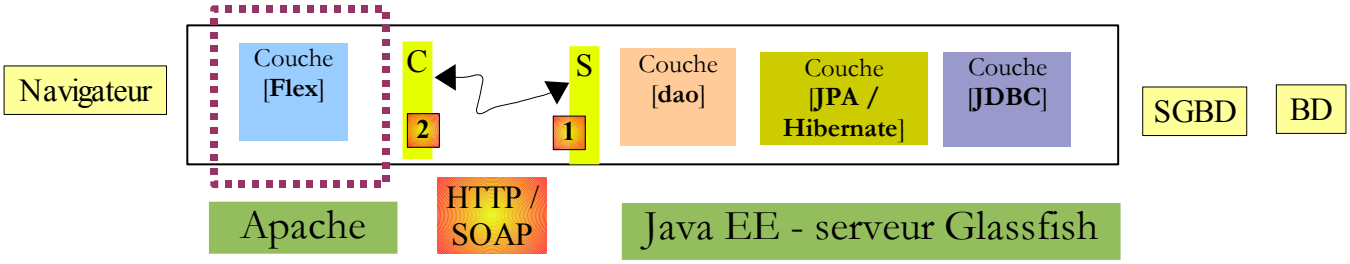
- l'exécution du client Flex se fait en local sur le navigateur sauf lorsqu'il a besoin de données externes. Dans ce cas, il les demande au serveur [3]. Il les reçoit en [4] selon des formats divers : XML ou binaire. L'application interrogée sur le serveur web peut être écrite dans un langage quelconque. Seul compte le format de la réponse.

Nous avons décrit l'architecture d'exécution d'une application Flex afin que le lecteur perçoive bien la différence entre celle-ci et celle d'une application Web classique, sans Ajax, telle que l'application Asp.Net décrite précédemment. Dans cette dernière, le navigateur est passif : il affiche simplement des pages Html construites sur le serveur web qui les lui envoie.

Dans la suite, nous donnons deux exemples de clients Flex dans le seul but de montrer la diversité des clients d'un service web. L'auteur étant lui-même un débutant Flex, certains points ne seront peut-être pas détaillés comme ils le devraient.

6.1 Un premier client Flex

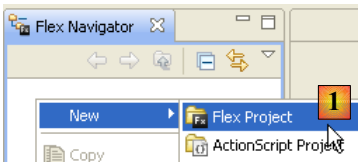
Nous écrivons maintenant un premier client Flex pour afficher la liste des clients. L'architecture client / serveur mise en place sera la suivante :

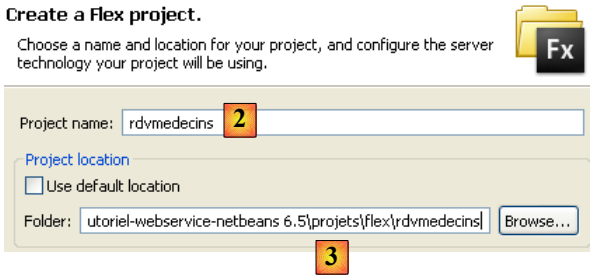


Dans cette architecture, il y a deux serveurs web :

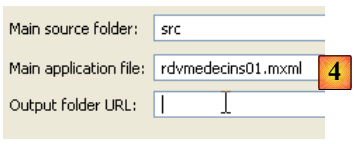
- le serveur Glassfish qui exécute le service web distant
- le serveur Apache qui exécute le client Flex du service web distant

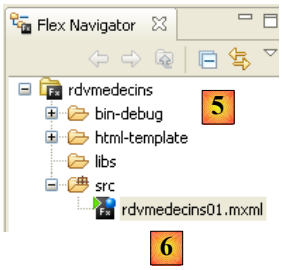
Nous construisons le client Flex avec l'IDE Flex Builder 3 :

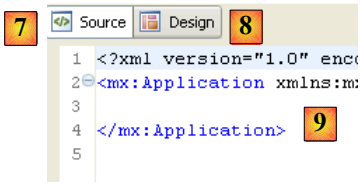




- dans Flex Builder 3, on crée un nouveau projet en [1]
- on lui donne un nom en [2] et on précise en [3] dans quel dossier le générer

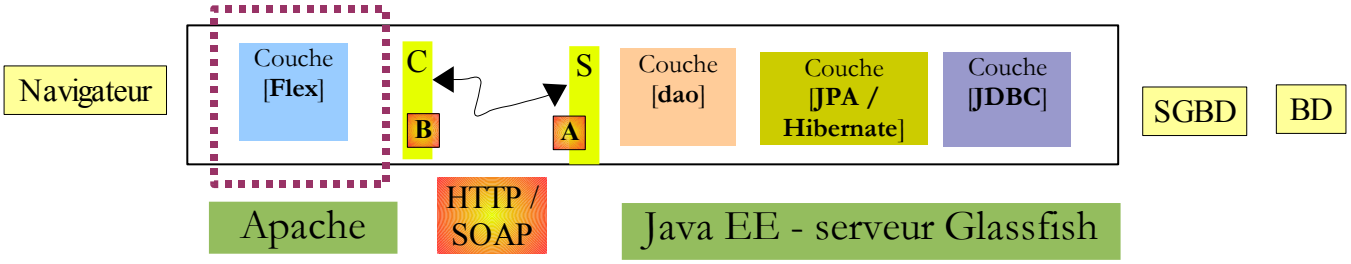




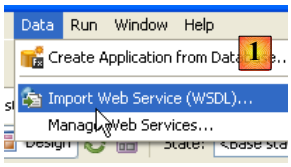


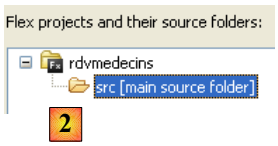
- en [4], on donne un nom à l'application principale (celle qui va être exécutée)
- en [5], le projet une fois généré
- en [6], le fichier MXML principal de l'application
- un fichier MXML comporte une vue et le code de gestion des événements de cette vue. L'onglet [Source] [7] donne accès au fichier MXML. On y trouvera des balises <mx> décrivant la vue ainsi que du code ActionScript.
- la vue peut être construite graphiquement en utilisant l'onglet [Design] [8]. Les balises MXML décrivant la vue sont alors générées automatiquement dans l'onglet [Source]. L'inverse est vrai : les balises MXML ajoutées directement dans l'onglet [Source] sont reflétées graphiquement dans l'onglet [Design].

Comme il a été fait avec les clients C# et Asp.Net précédents, nous allons générer le proxy local C [B] du service web distant S [A] :

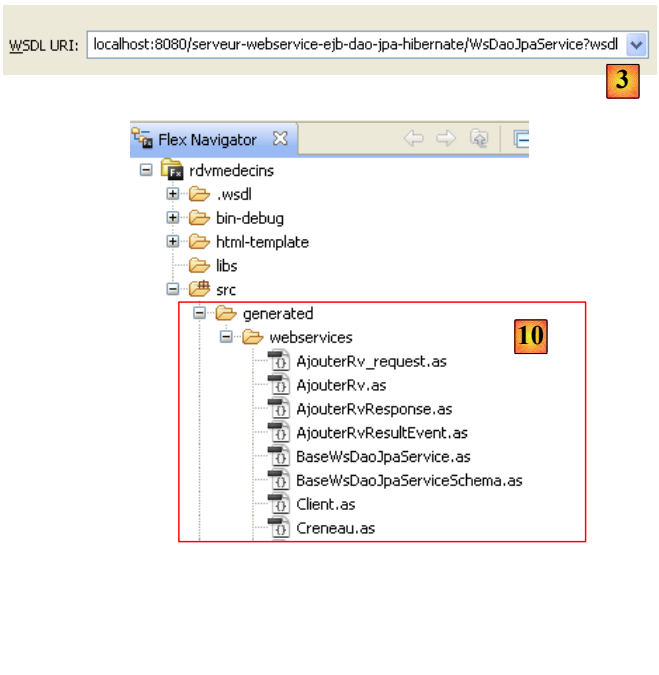


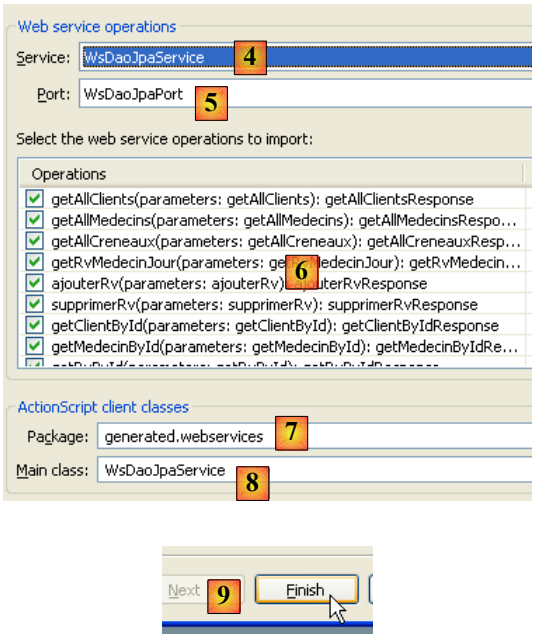
Pour que le proxy C puisse être généré, il faut que le service web JEE soit actif.



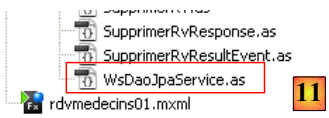


- en [1], choisir l'option *Data / Import Web Service*
- en [2], sélectionner le dossier de génération des classes et interfaces du proxy C.





- en [3], mettre l'Uri du fichier WSDL du service web distant S (cf paragraphe 4.10.2, page 46) puis passer à l'étape suivante
- en [4] et [5], le service web décrit par le fichier WSDL indiqué en [3]
- en [6] : la liste des méthodes qui vont être générées pour le proxy C. On notera que ce ne sont pas les méthodes réelles du service S. Elles n'ont pas la bonne signature. Ici, chaque méthode présentée a un unique paramètre quelque soit le nombre de paramètres de la méthode réelle du service web. Cet unique paramètre est une instance de classe encapsulant dans ses champs les paramètres attendus par la méthode distante.
- en [7] : le package dans lequel les classes et interfaces du proxy C vont être générées
- en [8] : le nom de la classe locale qui fera office de proxy vers le service web distant
- en [9] : terminer l'assistant.
- en [10] : la liste des classes et interfaces du proxy C généré.



- en [11] : la classe [WsDaoJpaService] implémentant les méthodes du proxy C.

La classe [WsDaoJpaService] générée implémente l'interface [IWsDaoJpaService] suivante :

```
1.  /**
2.   * Service.as
3.   * This file was auto-generated from WSDL by the Apache Axis2 generator modified by Adobe
4.   * Any change made to this file will be overwritten when the code is re-generated.
5.   */
6.  package generated.webservices{
7.     import mx.rpc.AsyncToken;
8.     import flash.utils.ByteArray;
9.     import mx.rpc.soap.types.*;
10.
11.     public interface IWsDaoJpaService
12.     {
13.       //Stub functions for the getAllClients operation
14.       /**
15.        * Call the operation on the server passing in the arguments defined in the WSDL file
16.        * @param getAllClients
17.        * @return An AsyncToken
18.        */
19.       function getAllClients(getAllClients:GetAllClients):AsyncToken;
20. ....
21.         function getAllClients_send():AsyncToken;
22. ...
23.         function get getAllClients_lastResult():GetAllClientsResponse;
24. ...
25.         function set getAllClients_lastResult(lastResult:GetAllClientsResponse):void;
26. ...
27.        function addgetAllClientsEventListener(listener:Function):void;
28. ...
29.         function get getAllClients_request_var():GetAllClients_request;
30. ...
31.         function set getAllClients_request_var(request:GetAllClients_request):void;
32. ...
33.    }
34. }
```

- ligne 11 : l'interface [IWsDaoJpaService] implémentée par la classe [WsDaoJpaService]
- lignes 19-31 : les différentes méthodes générées pour la méthode **getAllClients**() du service web distant. La seule qui se rapproche de celle réellement exposée par le service web est celle de la ligne 19. Elle porte le bon nom mais n'a pas la bonne signature : la méthode **getAllClients**() du service web distant n'a pas de paramètres.

Le paramètre unique de la méthode **getAllClients** du proxy C généré est de type **GetAllClients** suivant :

```
1.  /**
2.   * GetAllClients.as
3.   * This file was auto-generated from WSDL by the Apache Axis2 generator modified by Adobe
4.   * Any change made to this file will be overwritten when the code is re-generated.
5.   */
6.
7.  package generated.webservices
8.  {
9.     import mx.utils.ObjectProxy;
10.    import flash.utils.ByteArray;
11.    import mx.rpc.soap.types.*;
12.    /**
13.     * Wrapper class for a operation required type
14.     */
15.
16.    public class GetAllClients
17.    {
18.       /**
19.        * Constructor, initializes the type class
20.        */
```

```
21.       public function GetAllClients() {}
22.
23.    }
24. }
```

C'est une classe vide. Cela pourrait correspondre au fait que la méthode cible *getAllClients* n'admet pas de paramètre.

Maintenant examinons les classes générées pour les entités *Medecin*, *Client*, *Rv* et *Creneau*. Examinons par exemple la classe *Client* :

```
1.  package generated.webservices
2.  {
3.     import mx.utils.ObjectProxy;
4.     import flash.utils.ByteArray;
5.     import mx.rpc.soap.types.*;
6.
7.     public class Client extends generated.webservices.Personne
8.     {
9.        public function Client() {}
10.
11.    }
12. }
```

La classe *Client* est vide également. Elle dérive (ligne 7) de la classe *Personne* suivante :

```
1.  package generated.webservices
2.  {
3.     import mx.utils.ObjectProxy;
4.     import flash.utils.ByteArray;
5.     import mx.rpc.soap.types.*;
6.
7.     public class Personne
8.     {
9.        public function Personne() {}
10.
11.       public var id:Number;
12.       public var nom:String;
13.       public var prenom:String;
14.       public var titre:String;
15.       public var version:Number;
16.    }
17. }
```

- lignes 11-15 : on retrouve les attributs de la classe *Personne* définie au sein du service web JEE.

Nous avons les principaux éléments du proxy C. Nous pouvons désormais l'utiliser.

Le fichier principal [rdvmedecins01.xml] du client est le suivant :

```
1.  <?xml version="1.0" encoding="utf-8"?>
2.  <mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="vertical"
    creationComplete="init();">
3.     <mx:Script>
4.        <![CDATA[
5.           import generated.webservices.Client;
6.  ...
7.
8.           // données
9.           private var ws:WsDaoJpaService;
10.          [Bindable]
11.          private var clients:ArrayCollection;
12.
13.          private function init():void{
14. ...
15.          }
16.
17.          private function loadClients():void{
18. ...
19.          }
20.
21. ...
22.          private function displayClient(client:Client):String{
23. ...
24.          }
25.       ]]>
26.    </mx:Script>
```

```
27.    <mx:Label text="Liste des clients" fontSize="14"/>
28.    <mx:List dataProvider="{clients}" labelFunction="displayClient"></mx:List>
29.    <mx:Button label="Afficher les clients" click="loadClients()"/>
30.    <mx:Text id="txtMsgErreur" width="454" height="75"/>
31.
32. </mx:Application>
```

Dans ce code, il faut distinguer diverses choses :

- la définition de l'application (ligne 2)
- la description de la vue de celle-ci (lignes 27-30)
- les gestionnaires d'événements en langage ActionScript au sein de la balise <mx:Script> (lignes 3-26).

Commentons pour commencer, la définition de l'application elle-même et la description de sa vue :

- ligne 2 : définit
  - le mode de disposition des composants dans le conteneur de la vue. L'attribut *layout="vertical"* indique que les composants seront les uns sous les autres.
  - la méthode à exécuter lorsque la vue aura été instanciée, c.a.d. le moment où tous ses composants auront été instanciés. L'attribut *creationComplete="init();"* indique que c'est la méthode *init* de la ligne 13 qui doit être exécutée. *creationComplete* est l'un des événements que peut émettre la classe *Application*.
- les ligne 27-30 définissent les composants de la vue
- ligne 27 : définit un texte
- ligne 28 : une liste dans laquelle on mettra la liste des clients. La balise *dataProvider="{clients}"* indique la source des données qui doivent remplir la liste. Ici, la liste sera remplie avec l'objet *clients* défini ligne 11. Pour pouvoir écrire *dataProvider="{clients}"*, il faut que le champ *clients* ait l'attribut [Bindable] (ligne 10). Cet attribut permet à une variable ActionScript d'être référencée à l'extérieur de la balise <mx:Script>. Le champ *clients* est de type *ArrayCollection*, un type ActionScript qui permet de stocker des listes d'objets, ici une liste d'objet de type *Client*.
- ligne 29 : un bouton. Son événement *click* est géré. L'attribut *click="loadClients()"* indique que la méthode *loadClients* de la ligne 17 doit être exécutée lors d'un clic sur le bouton. Ce sera ce bouton qui déclenchera la demande au service web de la liste des clients.
- ligne 30 : une zone de texte destinée à afficher un éventuel message d'erreur qui serait renvoyé par le serveur sur la demande précédente.

Les lignes 27-30 génèrent la vue suivante dans l'onglet [Design] :

- [1] : a été généré par le composant *Label* de la ligne 27
- [2] : a été généré par le composant *List* de la ligne 28
- [3] : a été généré par le composant *Button* de la ligne 29
- [4] : a été généré par le composant *Text* de la ligne 30
- [5] : un exemple d'exécution

Examinons maintenant le code ActionScript de la page. Ce code gère les événements de la vue.

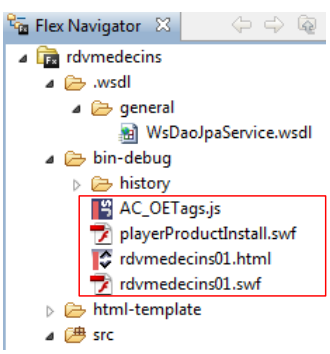
```
1.  <mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="vertical"
    creationComplete="init();">
2.
3.     <mx:Script>
4.        <![CDATA[
5.           import generated.webservices.Client;
6.  ...
7.
8.           // données
9.           private var ws:WsDaoJpaService;
10.          [Bindable]
11.          private var clients:ArrayCollection;
12.
13.          private function init():void{
14.             // on instancie le proxy du service web
15.             ws=new WsDaoJpaService();
16.             // on configure les gestionnaires d'évts
17.             ws.addgetAllClientsEventListener(loadClientsCompleted);
18.             ws.addEventListener(FaultEvent.FAULT,loadClientsFault);
19.          }
20.
21.          private function loadClients():void{
22.             // on demande la liste des clients
23.             ws.getAllClients(new GetAllClients());
24.          }
25.
26.          private function loadClientsCompleted(event:GetAllClientsResultEvent):void{
27.             // on récupère les clients dans le résultat envoyé
28.             clients=event.result as ArrayCollection;
29.          }
30.
31.          private function loadClientsFault(event:FaultEvent):void{
32.             // on affiche le msg d'erreur
33.             txtMsgErreur.text=event.fault.message;
34.          }
35.
36.          private function displayClient(client:Client):String{
37.             // on affiche un client
38.             return client.nom + " " + client.prenom;
39.          }
40.       ]]>
41.    </mx:Script>
42.    <mx:Label text="Liste des clients" fontSize="14"/>
43.    <mx:List dataProvider="{clients}" labelFunction="displayClient"></mx:List>
44.    <mx:Button label="Afficher les clients" click="loadClients()"/>
45. <mx:Text id="txtMsgErreur" width="454" height="75"/>
46.
```

- ligne 9 : *ws* va désigner le proxy C de type *WsDaoJpaService*, la classe générée précédemment qui implémente les méthodes d'accès au service web distant.
- ligne 13 : la méthode *init* exécutée lorsque la vue a été instanciée (ligne 1)
- ligne 15 : une instance du proxy C est créée
- ligne 17 : un gestionnaire d'événement est associé à l'événement "la méthode asynchrone *GetAllClients* s'est terminée avec succès". Pour toute méthode *m* du service web distant, le proxy C implémente une méthode *addmEventListener* qui permet d'associer un gestionnaire à l'événement "la méthode asynchrone *m* s'est terminée avec succès". Ici, la ligne 17 indique que la méthode *loadClientsCompleted* de la ligne 26 doit être exécutée lorsque le client Flex aura reçu la liste des clients.
- ligne 18 : un gestionnaire d'événement est associé à l'événement "une méthode asynchrone du proxy C s'est terminée sur un échec". Ici, la ligne 18 indique que la méthode *loadClientsFault* de la ligne 31 doit être exécutée à chaque fois qu'une requête asynchrone du proxy C vers le service web S échoue. Ici la seule requête qui sera faite est celle qui demande la liste des clients.
- au final, la méthode *init* de la ligne 13 a instancié le proxy C et défini des gestionnaires d'événements pour la requête asynchrone qui sera faite ultérieurement.
- ligne 21 : la méthode exécutée lors d'un clic sur le bouton [Afficher les clients] (ligne 44)
- ligne 23 : la méthode asynchrone *getAllClients* du proxy C est exécutée. On lui passe une instance *GetAllClients* chargée d'encapsuler les paramètres de la méthode distante appelée. Ici, il n'y a aucun paramètre. On crée une instance vide. La méthode *getAllClients* est asynchrone. L'exécution se poursuit sans attendre les données renvoyées par le serveur. L'utilisateur peut notamment continuer à interagir avec la vue. Les événements qu'il provoque continueront à être gérés. Grâce à la méthode *init*, nous savons que :
  - la méthode *loadClientsCompleted* (ligne 26) sera exécutée lorsque le client Flex aura reçu la liste des clients
  - la méthode *loadClientsFault* (ligne 31) sera exécutée si la requête se termine sur une erreur.

ligne 26 : la méthode *loadClientsCompleted* exécutée en cas de succès de la requête reçoit en paramètre un type *GetAllClientsResultEvent*. Pour chaque méthode *m* du service web distant, le proxy C définit une classe *MResultEvent* qui contient le résultat de la requête dans son champ *result*.

- ligne 28 : la liste des clients est récupérée dans l'événement. On sait que la méthode *getAllClients* du service web distant renvoie une liste. On met celle-ci dans le champ *clients* de la ligne 11. Un transtypage est nécessaire. La liste de la ligne 43 étant liée (Bindable) au champ clients, elle est avertie que ses données ont changé. Elle affiche alors la liste des clients. Elle va afficher chaque élément de la liste *clients* avec la méthode *displayClient* (ligne 43).
- ligne 36 : la méthode *displayClient* reçoit un type *Client*. Elle doit retourner la chaîne de caractères que doit afficher la liste pour ce client. Ici le nom et le prénom (ligne 38).
- ligne 31 : la méthode exécutée lorsqu'une requête au service web échoue. Elle reçoit un paramètre de type *FaultEvent*. Cette classe a un champ *fault* qui encapsule l'erreur renvoyée par le serveur. *fault.message* est le message accompagnant l'erreur.
- ligne 33 : le message d'erreur est affiché dans la zone de texte prévue à cet effet.

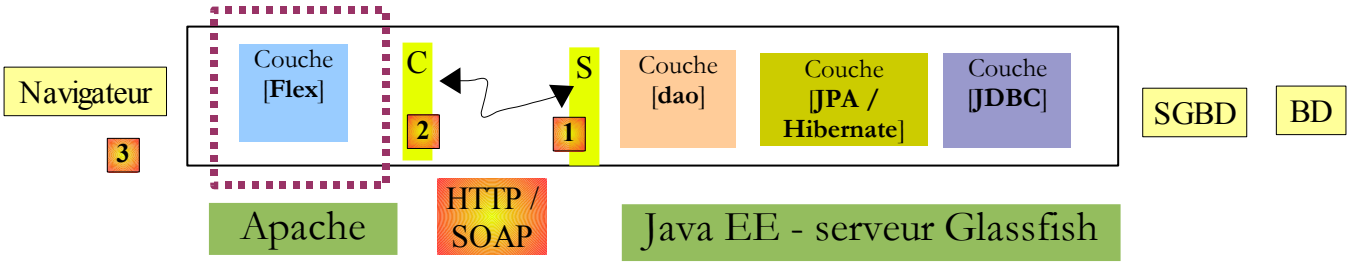
Lorsque l'application a été construite, son code exécutable se trouve dans le dossier [bin-debug] du projet Flex :



Ci-dessus,

- le fichier [rdvmedecins01.html] représente le fichier Html qui sera demandé par le navigateur au serveur web pour avoir le client Flex
- le fichier [rdvmedecins01.swf] est le binaire du client Flex qui sera encapsulé dans la page Html envoyée au navigateur puis exécuté par le plugin Flash Player de celui-ci.

Nous sommes prêts à exécuter le client Flex. Il nous faut auparavant mettre en place l'environnement d'exécution qui lui est nécessaire. Revenons à l'architecture client / serveur testée :

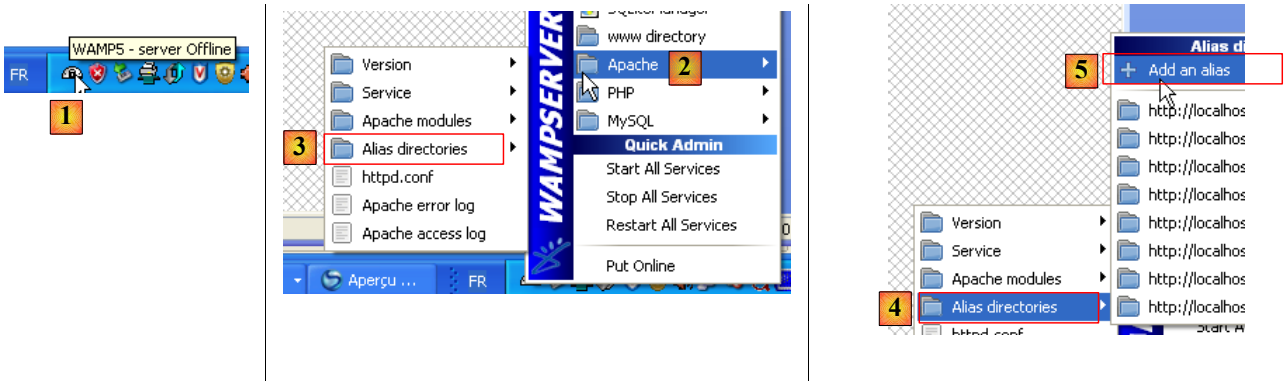


Côté serveur :

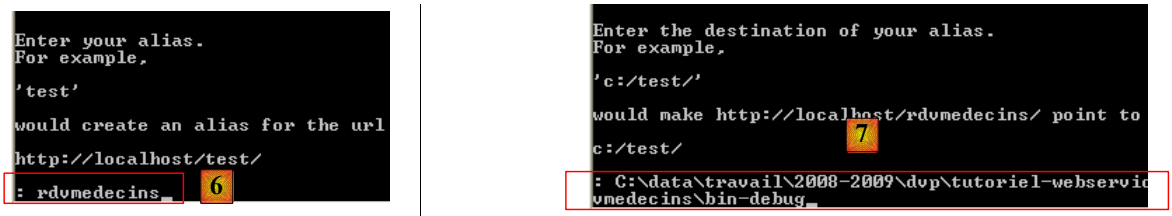
- lancer le SGBD MySQL
- lancer le serveur Glassfish
- déployer le service web JEE des rendez-vous s'il n'est pas déployé
- éventuellement tester l'un des clients précédents pour vérifier que tout est en place côté serveur.

Côté client :

Lancer le serveur Apache à qui sera demandée l'application Flex. Ici nous utilisons l'outil *Wamp*. Avec cet outil, nous pouvons associer un alias au dossier [bin-debug] du projet Flex.

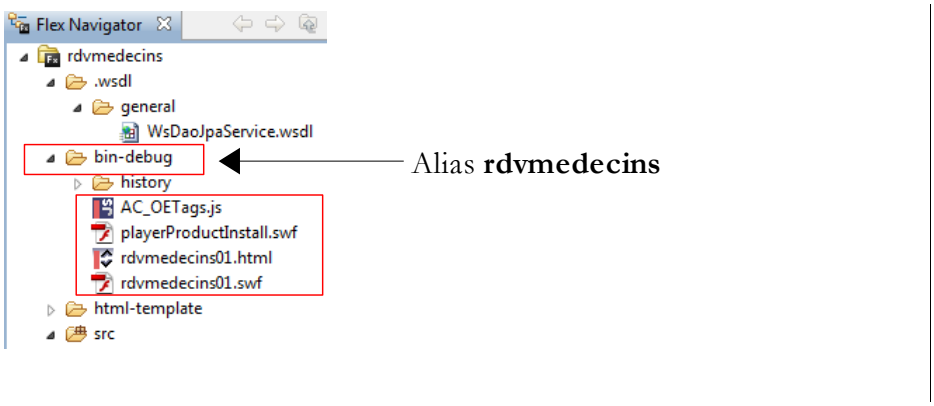


- l'icône de Wamp est en bas de l'écran [1]
- par un clic gauche sur l'icône *Wamp*, sélectionner l'option *Apache* [2] / *Alias Directories* [3, 4]
- sélectionner l'option [5] : *Add un alias*



- en [6] donner un alias (un nom quelconque) à l'application web qui va être exécutée
- en [7] indiquer la racine de l'application web qui portera cet alias : c'est le dossier [bin-debug] du projet Flex que nous venons de construire.

Rappelons la structure du dossier [bin-debug] du projet Flex :



Le fichier [rdvmedecins01.html] est le fichier Html de l'application Flex. Grâce à l'alias que nous venons de créer sur le dossier [bin-debug], ce fichier sera obtenu par l'url [http://localhost/rdvmedecins/rdvmedecins01.html]. Nous demandons celle-ci dans un navigateur disposant du plugin Flash Player version 9 ou supérieure :

- en [1], l'Url de l'application Flex
- en [2], nous demandons la liste des clients
- en [3], le résultat obtenu lorsque tout va bien
- en [4], le résultat obtenu lorsque nous demandons l'affichage des clients alors que le service web a été arrêté.

Vous pourriez avoir la curiosité d'afficher le code source de la page Html reçue :

```
1.  <!-- saved from url=(0014)about:internet -->
2.  <html lang="en">
3.
4.  <!--
5.  Smart developers always View Source.
6.
7.  This application was built using Adobe Flex, an open source framework
8.  for building rich Internet applications that get delivered via the
9.  Flash Player or to desktops via Adobe AIR.
10.
11. Learn more about Flex at http://flex.org
12. // -->
13.
14. <head>
15. <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
16.
17. <!--  BEGIN Browser History required section -->
18. <link rel="stylesheet" type="text/css" href="history/history.css" />
19. <!--  END Browser History required section -->
20.
21. <title></title>
22. <script src="AC_OETags.js" language="javascript"></script>
23. ...
24. <script language="JavaScript" type="text/javascript">
25. <!--
26. // -----------------------------------------------------------------------------
27. // Globals
28. // Major version of Flash required
29. var requiredMajorVersion = 9;
30. // Minor version of Flash required
31. var requiredMinorVersion = 0;
32. // Minor version of Flash required
33. var requiredRevision = 124;
34. // -----------------------------------------------------------------------------
35. // -->
36. </script>
37. </head>
38.
39. <body scroll="no">
40. <script language="JavaScript" type="text/javascript">
41. <!--
42. // Version check for the Flash Player that has the ability to start Player Product Install
    (6.0r65)
43. ....
44. // -->
45. </script>
```

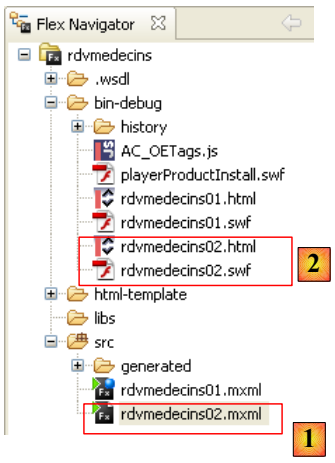
```
46. <noscript>
47.         <object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
48.                         id="rdvmedecins01" width="100%" height="100%"
49.                         codebase="http://fpdownload.macromedia.com/get/flashplayer/current/swflash
    .cab">
50.                         <param name="movie" value="rdvmedecins01.swf" />
51.                         <param name="quality" value="high" />
52.                         <param name="bgcolor" value="#869ca7" />
53.                         <param name="allowScriptAccess" value="sameDomain" />
54.                         <embed src="rdvmedecins01.swf" quality="high" bgcolor="#869ca7"
55.                                 width="100%" height="100%" name="rdvmedecins01" align="middle"
56.                                 play="true"
57.                                 loop="false"
58.                                 quality="high"
59.                                 allowScriptAccess="sameDomain"
60.                                 type="application/x-shockwave-flash"
61.                                 pluginspage="http://www.adobe.com/go/getflashplayer">
62.                         </embed>
63.         </object>
64. </noscript>
65. </body>
66. </html>
```

Le corps de la page commence ligne 39. Il ne contient pas du Html classique mais un objet (ligne 47) de type "application/x-shockwave-flash" (ligne 60). C'est le fichier [rdvmedecins01.swf] (ligne 54) que l'on peut voir dans le dossier [bin-debug] du projet Flex. C'est un fichier de taille importante : 600 K environ pour ce simple exemple.

6.2 Un deuxième client Flex

Le deuxième client Flex ne va pas utiliser le proxy C généré pour le premier. On veut montrer que cette étape n'est pas indispensable même si elle présente des avantages vis à vis de celle qui va être présentée ici.

Le projet évolue de la façon suivante :



- en [1] la nouvelle application Flex
- en [2] les exécutables qui lui sont associés
- en [3] la nouvelle vue : on va afficher la liste des médecins.

Le code MXML de l'application [rdvmedecins02.mxml] est le suivant :

```
1.  <?xml version="1.0" encoding="utf-8"?>
2.  <mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="vertical">
3.     <mx:Script>
4.        <![CDATA[
5.           import mx.rpc.events.ResultEvent;
6.           import mx.rpc.events.FaultEvent;
```

```
7.          import mx.collections.ArrayCollection;
8.
9.          // données
10.         [Bindable]
11.         private var medecins:ArrayCollection;
12.
13.         private function loadMedecins():void{
14.            // on demande la liste des médecins
15.            wsrdvmedecins.getAllMedecins.send();
16.         }
17.
18.         private function loadMedecinsCompleted(event:ResultEvent):void{
19.            // on récupère les médecins
20.            medecins=event.result as ArrayCollection;
21.         }
22.
23.         private function loadMedecinsFault(event:FaultEvent):void{
24.            // on affiche le msg d'erreur
25.            txtMsgErreur.text=event.fault.message;
26.         }
27.
28.         // affichage d'un médecin
29.         private function displayMedecin(medecin:Object):String{
30.            return medecin.nom + " " + medecin.prenom;
31.         }
32.
33.      ]]>
34.   </mx:Script>
35.   <mx:WebService id="wsrdvmedecins"
36.      wsdl="http://localhost:8080/serveur-webservice-ejb-dao-jpa-hibernate/WsDaoJpaService?wsdl">
37.      <mx:operation name="getAllMedecins"
38.         result="loadMedecinsCompleted(event)" fault="loadMedecinsFault(event);">
39.         <mx:request/>
40.      </mx:operation>
41.   </mx:WebService>
42.   <mx:Label text="Liste des médecins" fontSize="14"/>
43.   <mx:List dataProvider="{medecins}" labelFunction="displayMedecin"></mx:List>
44.   <mx:Button label="Afficher les médecins" click="loadMedecins()"/>
45.   <mx:Text id="txtMsgErreur" width="300" height="113"/>
46.
47. </mx:Application>
```

Nous ne commenterons que les nouveautés :

- lignes 42 - 45 : la nouvelle vue. Elle est identique à la précédente si ce n'est qu'elle a été adaptée pour afficher les médecins plutôt que les clients.
- lignes 35-41 : le service web est ici décrit par une balise <**mx:WebService**> (ligne 35). Le proxy C utilisé dans la version précédente n'est plus utilisé ici.
- ligne 35 : l'attribut *id* donne un nom au service web.
- ligne 36 : l'attribut *wsdl* donne l'uri du fichier WSDL du service web. C'est la même Uri qu'utilisée par le précédent client et définie page 46.
- lignes 37-40 : définissent une méthode du service web distant au moyen de la balise <*mx:operation*>
- ligne 37 : la méthode référencée est définie par l'attribut *name*. Ici nous référençons la méthode distante *getAllMedecins*.
- ligne 38 : on définit les méthodes à exécuter en cas de succès de l'opération (attribut *result*) et en cas d'échec (attribut *fault*).
- ligne 39 : la balise <*mx:request*> sert à définir les paramètres de l'opération. Ici la méthode distante *getAllMedecins* n'a pas de paramètre, donc nous ne mettons rien. Pour une méthode admettant des paramètres *param1* et *param2*, on écrirait :

```
<mx:Request>
  <param1>{param1}</param1>
  <param1>{param1}</param1>
</mx:Request>
```

où *param1* et *param2* seraient des variables déclarées et initialisées dans la balise <mx:Script>

```
[Bindable]
private var param1:Type1;
[Bindable]
private var param2:Type2;
```

Dans la balise <*mx:Script*> on trouve du code ActionScript analogue à celui étudié dans le client précédent. Seule diffère la méthode *loadMedecins* des lignes 13-16. C'est le mode d'appel de la méthode distante [getAllMedecins] qui diffère :

- ligne 15 : on utilise le service web [wsrdvmedecins] défini ligne 35 et son opération [getAllMedecins] définie ligne 37. Pour exécuter cette opération, la méthode **send** est utilisée. C'est elle qui démarre l'appel asynchrone de la méthode *getAllMedecins* du service web défini ligne 35. La méthode *send* fera l'appel avec les paramètres définis par la balise <*mx:request*> de la ligne 39. Ici il n'y a aucun paramètre. Si la méthode avait eu les paramètres *param1* et *param2*, le script *loadMedecins* aurait affecté des valeurs à ces paramètres avant d'appeler la méthode *send*.

Il ne nous reste plus qu'à tester cette nouvelle application :

7 Conclusion

Nous avons montré les démarches qui mènent à la création et au déploiement d'un service web JEE avec l'IDE Netbeans 6.5 et le serveur d'application Glassfish qui l'accompagne. Par ailleurs, nous avons présenté divers clients pour ce service web : clients Java, C#, Asp.Net, Flex. C'est cette diversité des clients qui fait l'intérêt des services web.

Le lecteur intéressé à approfondir le sujet des services web JEE pourra lire les divers tutoriels disponibles sur le site de Netbeans : [http://www.netbeans.org/kb/trails/java-ee.html].

Au paragraphe 3, page 3, nous avons présenté des copies d'écran d'un client web Asp.net. Celui-ci fait l'objet d'un exercice d'université qu'on trouvera à l'Url [http://tahe.ftp-developpez.com/fichiers-archive/dotnet/exercices/rdvmedecins.zip].

Table des matières