**Tomcat :**

***Les répertoires tomcat :***

* /bin : contient le bash pour le démarrage du tomcat, exactement **startup.bat**
* /lib : contient les bibliothèques d’exécution de serveur tomcat (Servlet, JSP, ….)
* /log : contient les fichiers de journalisations de toutes les activités d’accès du tomcat.
* /temp : contient des fichiers temporaires nécessaire pour le fonctionnement du tomcat
* /webapps : répertoire utilisé pour le déploiement des application web, il faut juste copier-coller l’app dans ce répertoire, le cas où le serveur est á distant il faut passer par l’application manager du tomcat.
* /conf : dossier qui contient la configuration du tomcat

***Quelque fichier de configuration du répertoire /conf***

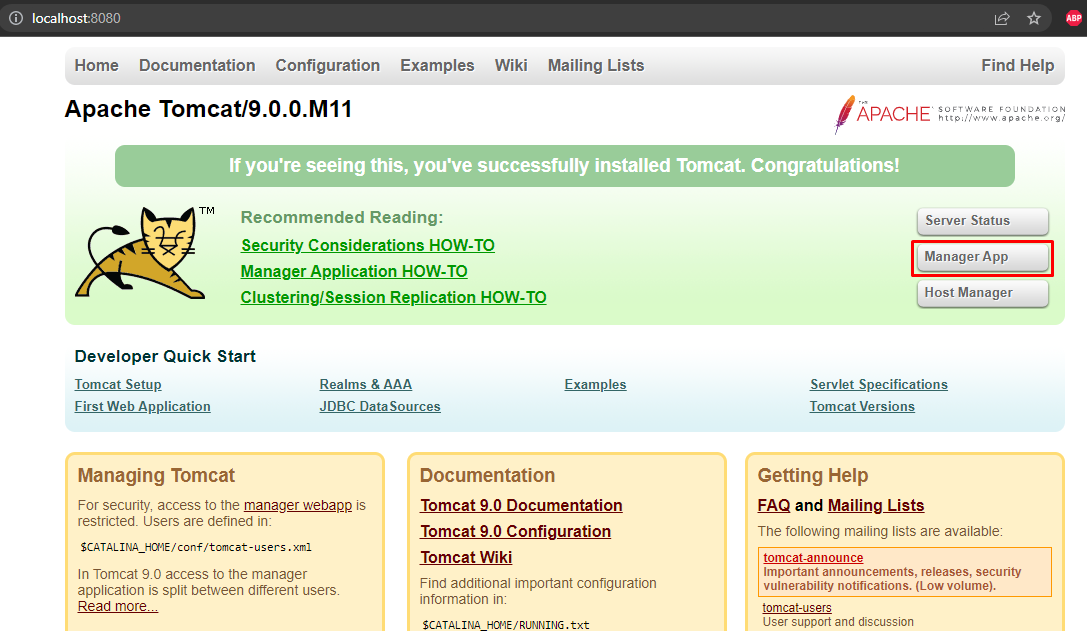
Serveur.xml : fichier principal par lequel tomcat démarre, contient

* <GlobalNamingResources /> : utilisé pour définir des ressources globales accessible par toutes les applications tournées sur le serveur, et contient aussi une ressource userdatabase qui est une base des utilisateurs qui gérer l’accès á distant de tomcat.
* <Connector /> : sert á changer le numéro du port d’écoute du tomcat.
* <Engine/> : utiliser pour référencier les services utilisé.
* <Host/> : { appBase : ‘’sert á indiquer le répertoire de déploiment des application’’ , unpackWARS : ‘’le fichier war sera décompreser’’, autoDeploy : ‘’auto déployer l’application dès sa copie dans le dossiers webapps’’}

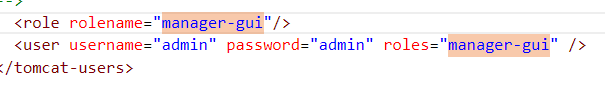
Logging.propreties : fichier principal de journalisation de tomcat

Tomcat\_users.xml : contient les utilisateurs et les différents rôles d’accès au serveur tomcat.

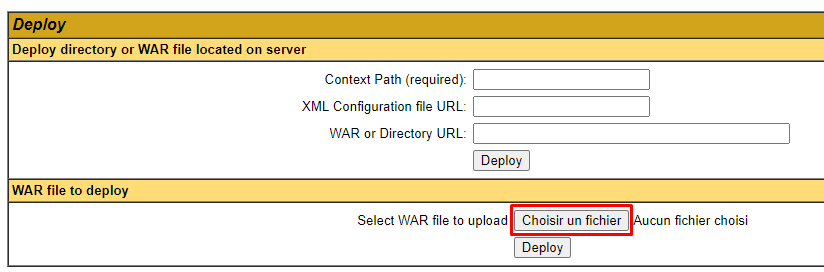
Déployer Projet web sous TOMCAT:

Tout d’abord il faut lancer tomcat á partir du fichier tomcat/bin/startup.bat, et d’accéder après l’application racine (root) du tomcat par le lient suivante : <http://localhost:8080/>

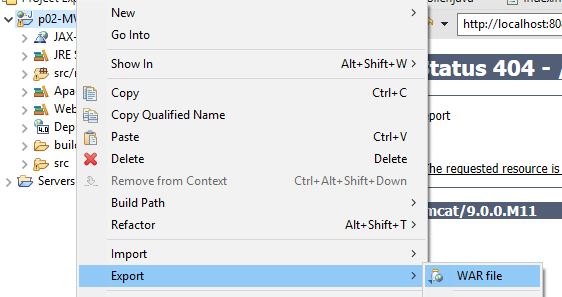
Alors, après il faut accéder á Manager App qui nécessite de se disposer d’un user qui a comme rôle manager-gui, si non il faut les créer dans les ajouter dans le fichier /conf/tomcat\_users.xml sous la balise tomcat\_users :



Après sous l’interface de manager á la section deploy, vous cliquer sur choisir un fichier et vous sélectionnez le fichier war de votre application.



Pour génerer le fichier war de votre projet vous devez cliquer sur button droit, export et finalment war file :



**Classe d’action :** la classe d’action permet de remplir le modèle á partir du couche métier, et d’appeler la vue pour l’affichage des donnés, le modèle est un objet de type Object afin de ne pas dépendre chaque fois d’un certain type.

**public** **class** BiblioAction {

**private** Object model;

**private** BiblioService service;

**public** BiblioAction() {

/\*Temporairement\*/

DataSource ds = **new** MySqlDataSource("Biblio");

Database db = **new** Database(ds);

AuthorDao authorDao = **new** AuthorDaoJdbc(db);

service = **new** BiblioServiceDefault(authorDao);

}

**public** Object getModel() {

**return** model;

}

/\*

\* Méthode d'action

\*/

**public** String authorsList() {

model = service.authors();

**return** "AuthorsList";

}

}

**Classe Controller : est** annoter par webServlet avec un path générique /docs/\* chaque requête commence par le lien /docs/ tomcat va la rediriger á cette contrôleur, la partie \* va définir l’action que le client vaudrait exécuter. Á l’aide de request.getRequestURI().endsWith(‘’action’’) on arrive á savoir l’action , il reste juste de l’appeler á partir d’un objet de la classe d’action pour finalement extraire la vue et du modèle convenable.

@WebServlet({ "/docs/\*", "\*.docs" })

**public** **class** BiblioController **extends** HttpServlet {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** BiblioAction action;

**private** String prefix = "/Views/";

**private** String suffix = ".jsp";

@Override

**public** **void** init() **throws** ServletException {

// init() permet d'acceder au context param, par contre dans le constructeur ce n'est pas possible

action = **new** BiblioAction();

//on peut faire le wiring ici

}

**protected** **void** doGet(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {

String view = "Error";

Object model = "Page Introuvable!";

PrintWriter out = response.getWriter();

//out.print("<h1>hello, world</h1");

/\*

\* ObjectOutput os = new ObjectOutputStream(response.getOutputStream());

\* os.writeObject(os);//communiquer avec un client lourd.(Applet)

\*/

**if**(request.getRequestURI().endsWith("/examples")) {

view = "printAuthor";

model = **new** Author(101, "James Gosling",1955);

}**else** **if**(request.getRequestURI().endsWith("/AuthorsList")) {

view = action.authorsList();

model = action.getModel();

}

request.setAttribute("model", model);

request.getRequestDispatcher(prefix + view + suffix).forward(request, response);

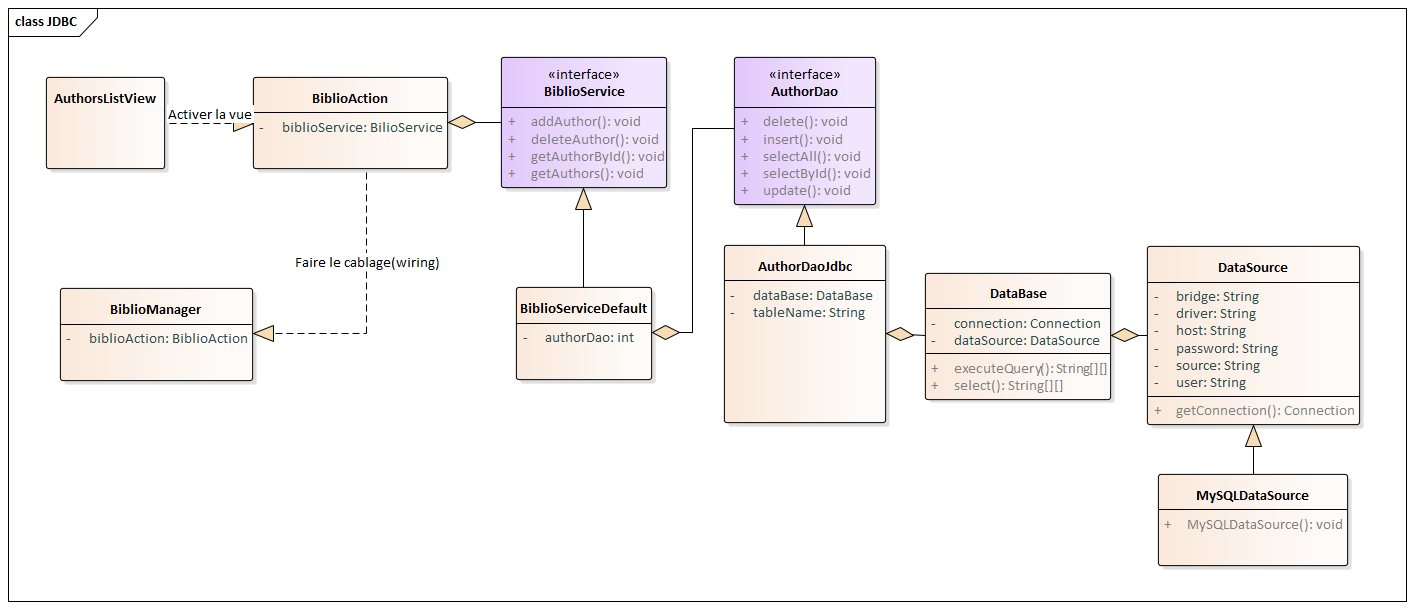
}

**protected** **void** doPost(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {

doGet(request, response);

}

}

La classe **DataSource** est une classe qui fournit la connexion vers toutes les bases de données relationnelles, ici par exemple pour accéder au serveur MySQL nous avons créés la classes MySQLDataSource qui étant de la classe DataSource et qui contient les informations du driver MySQL et le chemin vers le serveur.

**La classe DataSource**

package org.mql.java.jdbc;

import java.sql.Connection;

import java.sql.DriverManager;

public class DataSource {

private String driver ;

private String bridge;

private String host;

private String source;

private String user;

private String password;

public DataSource(String driver, String bridge, String host, String source, String user, String password) {

super();

this.driver = driver;

this.bridge = bridge;

this.host = host;

this.source = source;

this.user = user;

this.password = password;

}

public Connection getConnection() {

try {

Class.forName(driver);

String url = bridge + "//" + host + "/" + source;

Connection db = DriverManager.getConnection(url, user, password);

System.out.println("Connextion bien établie ...");

return db;

} catch (Exception e) {

System.out.println("Erreur : "+ e.getMessage());

return null;

}

}

}

Explication de quelque instruction de la classe précédente :

* Class.forName(driver) : permet de charger la classe par l’indication de son nom complétement qualifié, permet de charger le driver mysql au niveau de la mémoire.
* DriverManager.getConnection(url, user, password): retourne l’objet connection á partir de la classe DriverManager.

La classe MySQLDataSource :

**package** org.mql.java.jdbc;

**public** **class** MySqlDataSource **extends** DataSource {

**public** **static** **final** String ***MYSQL\_DRIVER*** = "com.mysql.jdbc.Driver";

**public** **static** **final** String ***MYSQL\_BRIDGE*** = "jdbc:mysql:";

**public** MySqlDataSource(String host, String source, String user, String password) {

**super**(***MYSQL\_DRIVER***, ***MYSQL\_DRIVER***, host, source, user, password);

}

**public** MySqlDataSource(String source) {

**super**(***MYSQL\_DRIVER***, ***MYSQL\_BRIDGE***, "localhost", source, "root", "");

}

**public** MySqlDataSource() {

**super**(***MYSQL\_DRIVER***, ***MYSQL\_BRIDGE***, "localhost", "mysql", "root", "");

}

}

La classe DataBase : est une classe où en interrogeant la base des données par les requêtes SQL :

package org.mql.java.jdbc;

import java.sql.Connection;

import java.sql.ResultSet;

import java.sql.ResultSetMetaData;

import java.sql.Statement;

//facade indepandante de toute type de base de données

//driver JDBC une applimentation java, pour la specification java.sql

public class Database {

private DataSource dataSource;

private Connection db;

public Database(DataSource ds) {

dataSource = ds;

db = ds.getConnection();

}

public String[][] executeQuery(String query) {

try {

Statement sql = db.createStatement();

ResultSet rs = sql.executeQuery(query);

//executeQuery pour select; retourne resultSet

//executeUpdate pour insert, update, delete retourne le nombre de lignes affecter

//execute() pour les requete DDL

//rs.next, rs.previous, rs.first, rs.last

//rs.beforeFirst, rs.afterLast

//

rs.last();

int rows = rs.getRow();

rs.beforeFirst();

ResultSetMetaData rsm = rs.getMetaData();/\*contient des information sur les ligne récupéré

le cas de select le nombre des colonnes \*/

//les colonnes se commencent par 1

int cols = rsm.getColumnCount();

String data[][] = new String[rows + 1][cols];

for(int i = 0; i < cols;i++) {

data[0][i] = rsm.getColumnName(i + 1);

}

int row = 0;

while(rs.next()) {

row++;

for (int i = 0; i < cols; i++) {

data[row][i] = rs.getString(i+1);

}

}

rs.close();

return data;

} catch(Exception e) {

System.out.println("Erreur + "+ e.getMessage());

return null;

}

}

public String[][] select(String tableName){

return executeQuery("SELECT \* FROM "+ tableName);

}

public String[][] selectLike(String tableName, String key, Object value){

return executeQuery("SELECT \* FROM "+ tableName + " WHERE "+ key + " LIKE '%" + value + "%'");

}

}

Explication de quelque instruction de la classe précédente :

Dans la methode executeQuery qui retourne une matrice des string, pour se connecter á la base de donnée il faut executer la requete avec depuis un objet Statment qui est instanciable depuis l’objet Connection. Cet le resultat de la requete est un Objet ResultSet qui ressemble dans son structure á une table sql :

* resultSet.next() : permet de parcourir le résultat ligne par ligne.
* resultSet.first() : Pointer sur la première ligne (Premier enregistrement)
* resultSet.beforeFirst() : pointer sur le nom des colonnes de la table sql.
* resultSet.last() : pointer sur le dernier enregistrement.
* resultSet.getRow() : retourne le numéro de la ligne courante.
* resultSet.getMetaData() : permet de savoir des informations concernant le résultat.
* resultSetMetaData.getColumnCount() : savoir combien de colonne.
* resultSetMetaData.getColumnName(int i) : savoir le nom de la colonne indice i.