Applications JavaEE Avancées

Développement d'applications plus orientées entreprise avec l'utilisation des Entreprise Java Beans, de la Java Persistence API et en utilisant le serveur d'application JBoss WildFly

- Serveur d'application Wildfly
- Enterprise Java Beans
- Persistance avec JPA
- WebServices JAX-RS

Serveur d'application Wildfly

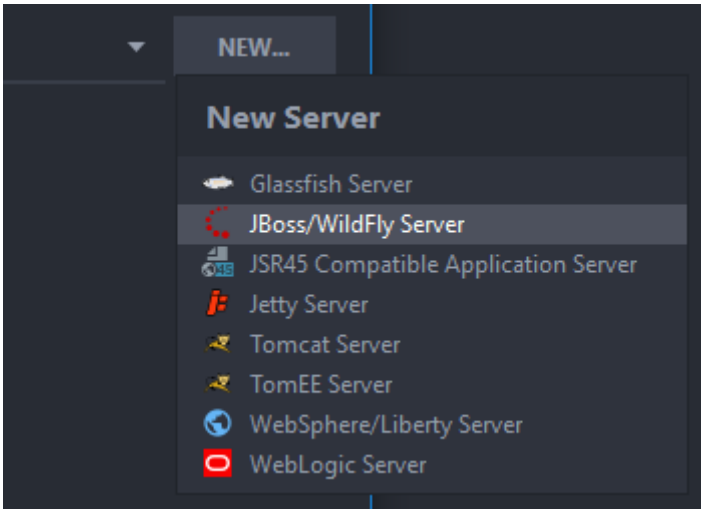
Jusqu'ici, nous n'avions besoin que d'un conteneur de servlet pour exécuter nos applications, c'est pourquoi Tomcat nous suffisait. Cependant, pour utiliser les spécifications JakartaEE que nous allons étudier dans cette partie, il faut disposer d'un serveur d'application JakartaEE complet. Nous allons ici utiliser Wildfly (évolution de JBoss).

Installation

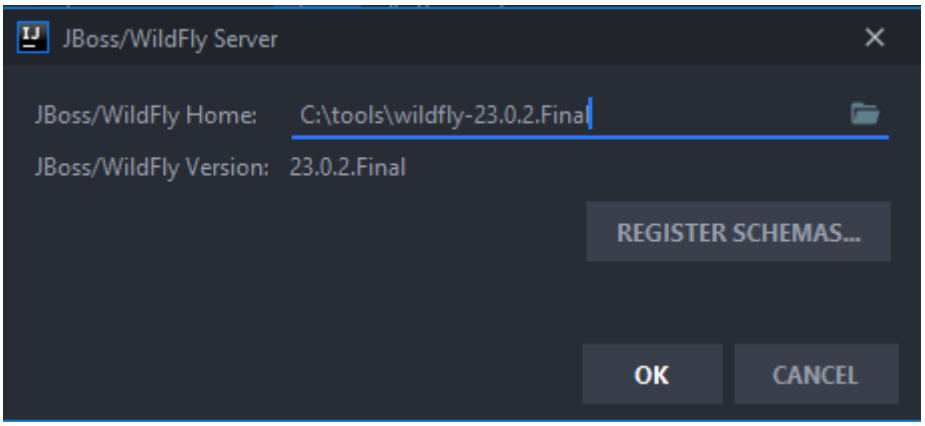
Commencez donc par télécharger Wildfly 23 sur ce lien. Décompressez ensuite le ZIP téléchargé et placez le à l'emplacement où vous rangez vos outils.

Créer un Projet avec WildFly

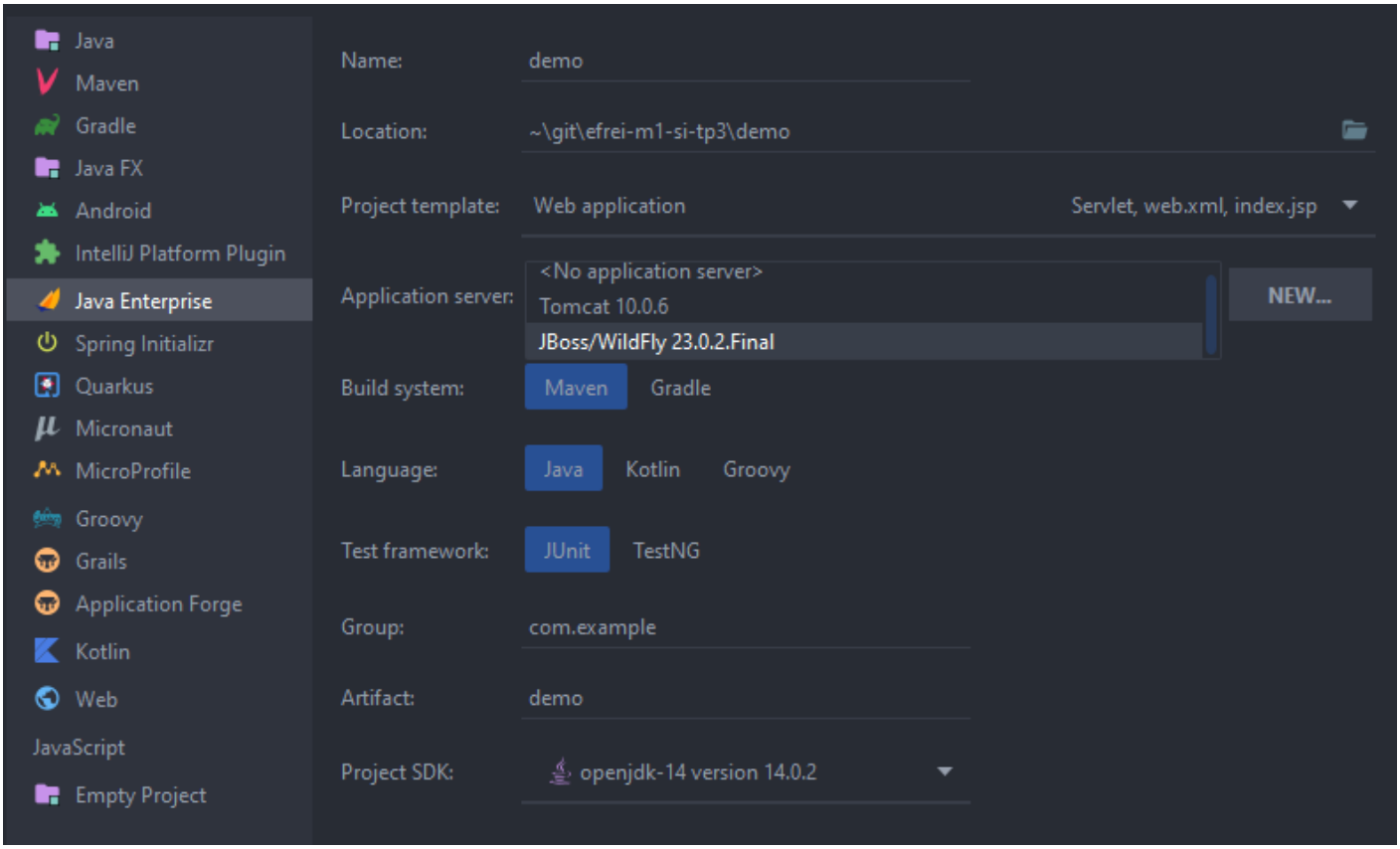
Pour créer un nouveau projet sur votre nouveau serveur Wildfly, utilisez le menu "New..." :



Indiquez ensuite le dossier d'installation de votre serveur :



Vous pouvez désormais le séléctionner dans le menu déroulant :



Enterprise Java Beans

Les EJB pour Enterprise Java Beans sont un standard de la pateforme JakartaEE pour créer des composants serveur. Cette architecture propose un conteneur pour créer des composants Java distribués hébergés sur un serveur d'application.

Il existe plusieurs types d'EJB.

Entity

Les EJB Entity permettent de représenter les données manipulées par l'application. On en parlera plus en détail dans le chapitre suivant sur JPA.

Message

Les EJB Message Driven servent à accomplir des tâches de manière asynchone en utilisant des mécanismes de file de message.

Session

Les EJB session permettent de proposer des services avec ou sans état. Il pour être injectés par le conteneur EJB à un service local (comme une servlet par exemple), mais il peut aussi être appelé par un client distant, via le protocole RMI (Remote Method Invocation).

EJB Session

Interface

Pour définir un EJB session, il faut définir une interface qui définir les méthodes accessibles par le code client. Il faut y ajouter une interface pour signaler au conteneur EJB qu'il s'agit d'une interface d'EJB ; `@Remote` pour un accès par un client RMI distant, et `@Local` pour un accès par un composant local déployé sur le serveur également.

```
@Remote
public interface HelloWorldRemote {
    String helloWorld(String name);
}
```

```
@Local
public interface HelloWorldLocal {
    String helloWorld(String name);
}
```

Implémentation

On peut ensuite créer une classe EJB qui implémente cette interface. On doit utiliser une annotation pour signaler au conteneur qu'il s'agit d'une classe d'implémentation EJB. On utilise `@Stateless` pour un composant qui sera instancié à chaque demande. Le paramètre d'annotation `mappedName` permet de préciser un nom dans l'annuaire JNDI qui va référencer tous les EJB sur serveur.

```
@Stateless(mappedName="HelloWorld")
public class HelloWorld implements HelloWorldRemote {
    public String helloWorld(String name) {
        return "Hello, " + name;
    }
}
```

Pour avoir toujours la même instance du composant fournie par le conteneur pour une session de client donnée, on peut utiliser à la place `@Stateful`.

```
@Stateful(mappedName="HelloWorld")
public class HelloWorld implements HelloWorldRemote {
    public String helloWorld(String name) {
        return "Hello, " + name;
    }
}
```

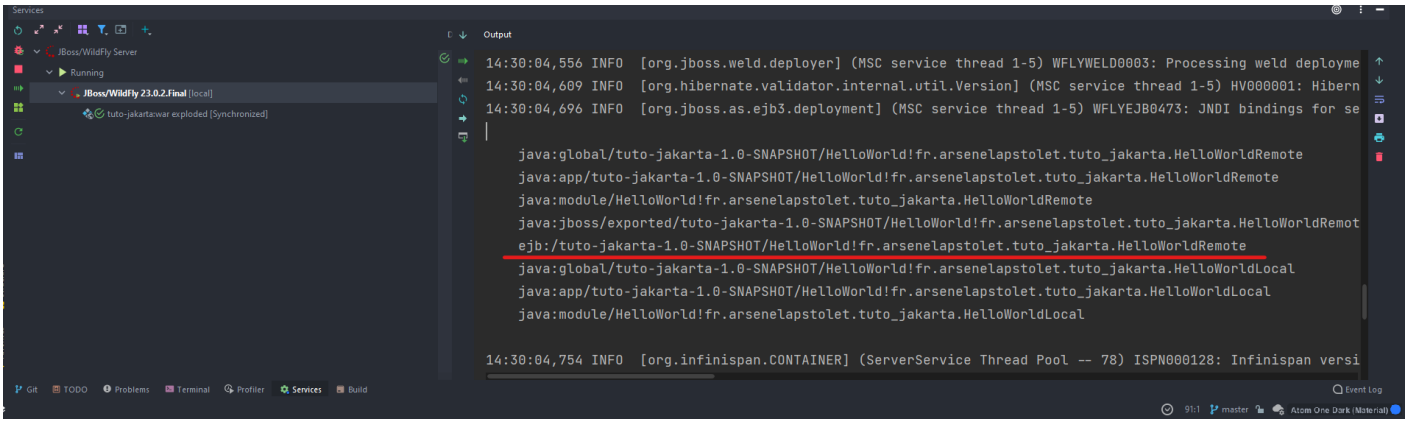
Utilisation

Pour demander au conteneur d'EJB d'injecter un EJB dans un autre composant, comme un autre EJB, ou une Servlet par exemple, utilisez l'annotation `@EJB` sur un attribut du type d'une interface EJB locale :

```
@EJB
private HelloWorldLocal helloWorld;
```

Client RMI

On peut accèder aux EJB remote via la protocole RMI depuis un client distant Java. Après avoir lancé le serveur Wildfly qui possède des EJB Remote, on peut regarder ensuite les logs du serveur Wildfly dans l'interface d'IntelliJ IDEA, et y voir une ligne affirmant que l'EJB a bien été detecté et ajouté à l'annuaire JNDI. Copiez l'URL qui commence par EJB et gardez là dans un bloc note, elle sera utile plus tard.



Configuration du projet Client RMI

Pour le client EJB, créez un nouveau projet IntelliJ, cette fois en utilisant le template **Maven** et en utilisant l'archétype **Quickstart**. Une fois votre projet généré, rajoutez la dépendance suivante dans votre fichier `pom.xml` :

```
<dependency>
  <groupId>org.wildfly</groupId>
  <artifactId>wildfly-ejb-client-bom</artifactId>
  <version>21.0.0.Final</version>
  <type>pom</type>
</dependency>
```

Mettez à jours vos dépendances Maven, puis ajoutez une Run Config d'application Java normale.

Récupérer les EJB

Dans un premier temps, il faut copier le package qui contient les interfaces dans le projet client (les classes doivent avoir le même nom et le même package dans le projet client). Retirez dans le projet client les annotations `@Remote` des interfaces.

Dans la méthode `main` ajoutez d'abord le code suivant, afin de configurer la connexion EJB :

```
final Hashtable<String, String> jndiProperties = new Hashtable<>();
jndiProperties.put(Context.INITIAL_CONTEXT_FACTORY,
"org.wildfly.naming.client.WildFlyInitialContextFactory");
if(Boolean.getBoolean("http")) {
  jndiProperties.put(Context.PROVIDER_URL,"http://localhost:8080/wildfly-services");
```

```
} else {
  jndiProperties.put(Context.PROVIDER_URL,"remote+http://localhost:8080");
}
final Context context = new InitialContext(jndiProperties);
```

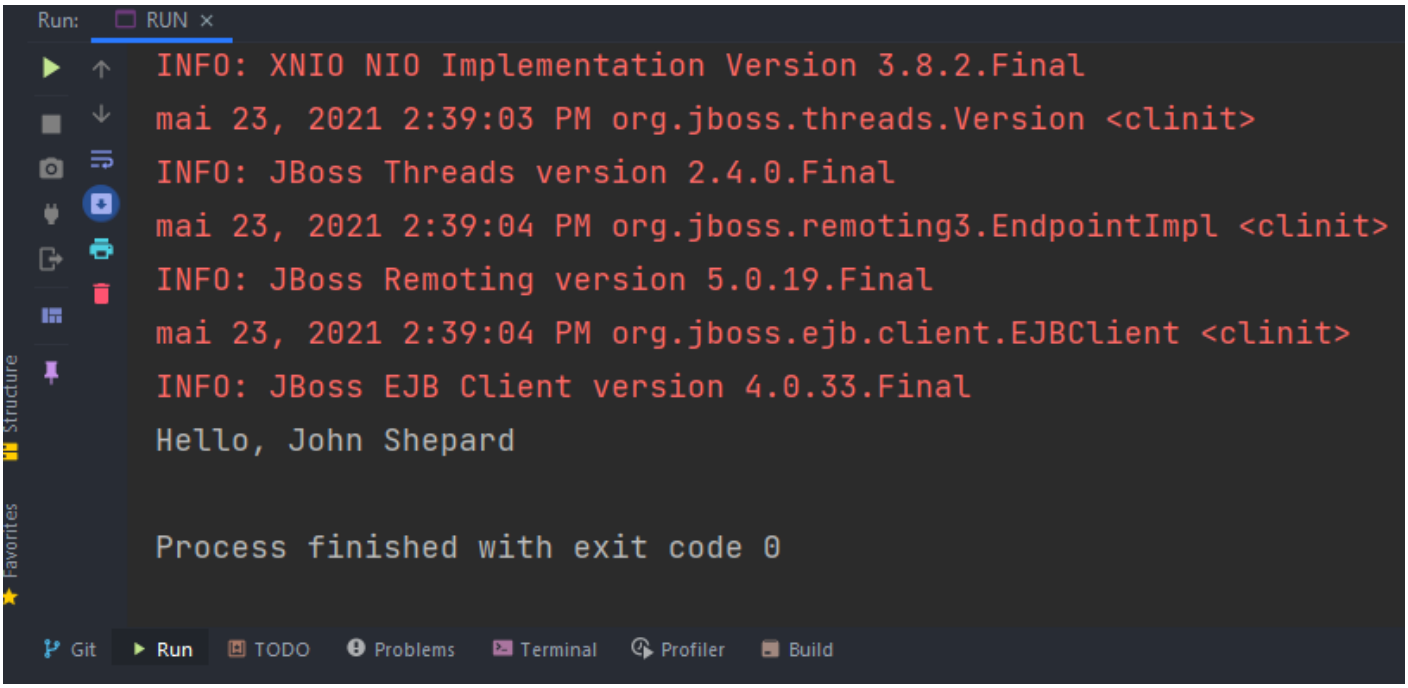
Ensuite pour récupérer votre EJB, ajouter ce code, en modifiant la string passée à la méthode `lookup()` par l'URL ejb que nous avons noté tout à l'heure :

```
final GestionContactRemote ejb = (GestionContactRemote) context.lookup("ejb:/tuto-jakarta-1.0-SNAPSHOT/HelloWorld!fr.arsenelapostolet.tuto_jakarta.HelloWorldRemote");
```

Vous pouvez enfin tester votre EJB :

```
System.out.println(ejb.helloWorld("John Shepard"));
```

Si tout s'est bien passé, vous devriez observer les logs du client EJB dans votre console ainsi que votre afichage :



Persistance avec JPA

JPA pour Java Persistance API est la spécification ORM de la plateforme JavaEE. Un ORM, pour Object Relationnal Mapper, est un composant logiciel chargé de faire le lien entre le modèle objet et le modèle relationnel et permet ainsi de persister des classes Java dans une système de gestion de base de donnée relationnel sans écrire tout le code JDBC directement. Le moteur vas ainsi établir et exécuter les opérations JDBC nécéssaires. JPA étant une spécification, on a besoin de choisir une implémentation pour l'utiliser. Nous allons utiliser la plus connue, Hibernate, mais il en existe d'autres, comme EclipseLink, TopLink, etc ...

Dépendances

Pour utiliser JPA avec Hibernate, il faut rajouter ces dépendances à notre projet :

```
<dependency>
  <groupId>jakarta.persistence</groupId>
  <artifactId>jakarta.persistence-api</artifactId>
  <version>2.2.3</version>
</dependency>

<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>5.4.22.Final</version>
</dependency>

<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-entitymanager</artifactId>
  <version>5.4.22.Final</version>
</dependency>
```

Data Source

Pour pouvoir accèder à une base donnée avec JPA, il faut configurer une Datasource sur notre serveur d'application. Pour cela, nous allons devoir accèder à l'interface d'administration de notre serveur Wilfly.

Créer un utilisateur d'Administration sur Wildfly

Rendez vous dans `WILDFLY_HOME/bin` et exécutez :

```
./add-user.bat
```

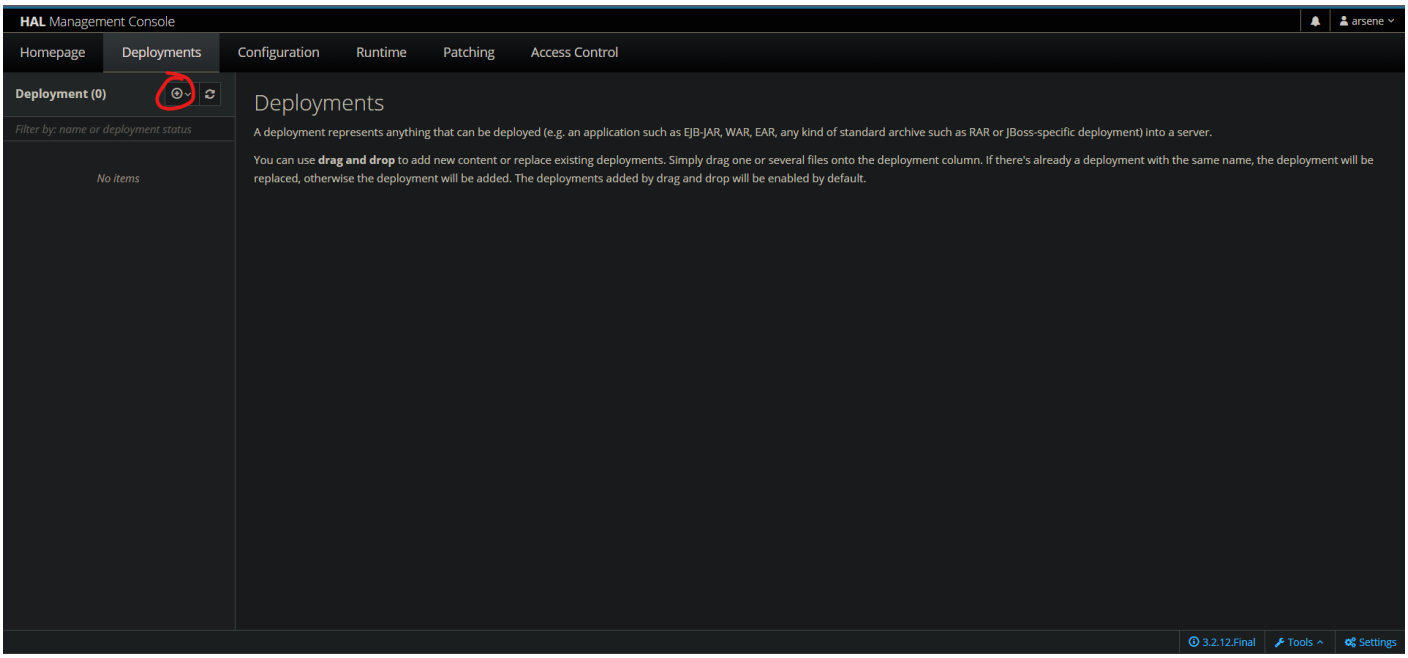
Suivez ensuite les instruction pour créér un utilisateur. Lorsqu'on l'on vous demande le type d'utilisateur choisissez **Utilisateur d'administration**.

Vous pouvez ensuite accèder au panneau d'administration Wildfly via ce lien (Le serveur Wildfly doit être démarré). Les identifiants pour se connecter sont ceux donnés à la création de l'utilisateur.

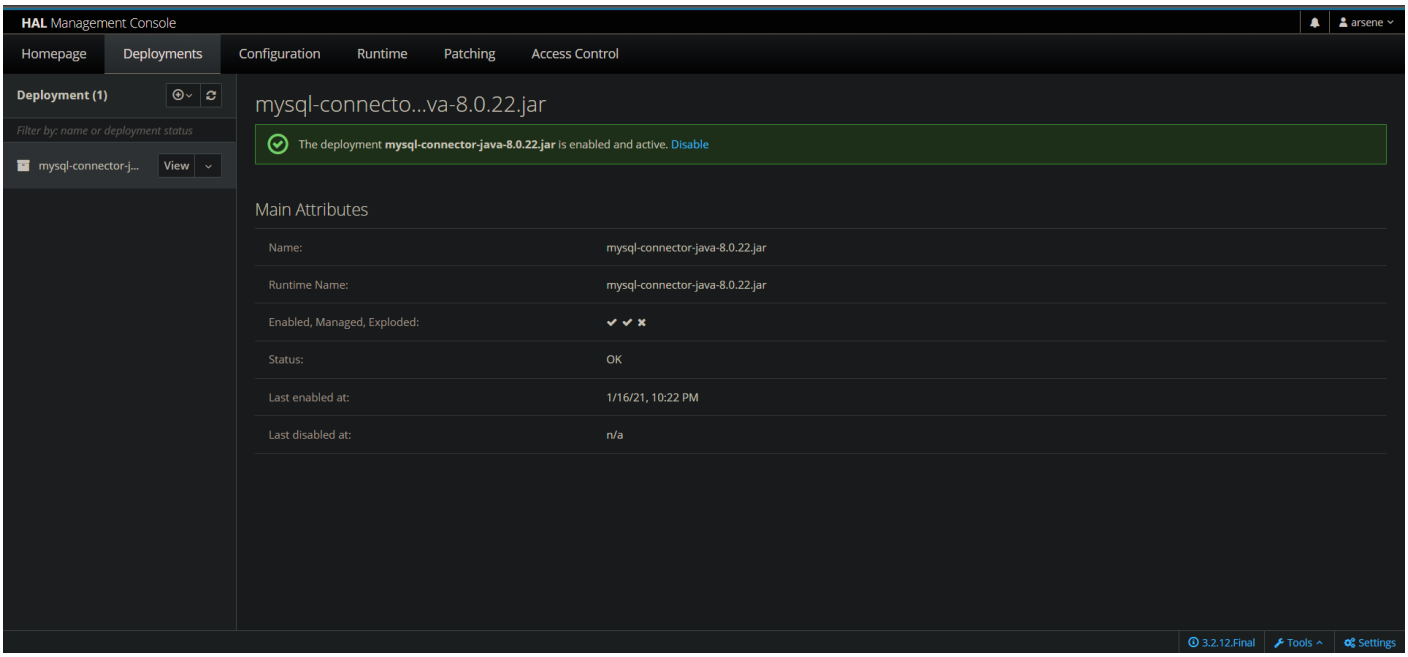
Configurer le Driver

Commencez par télécharger le Driver JDBC MySQL sur ce lien, décompressez le et récupérez le fichier JAR.

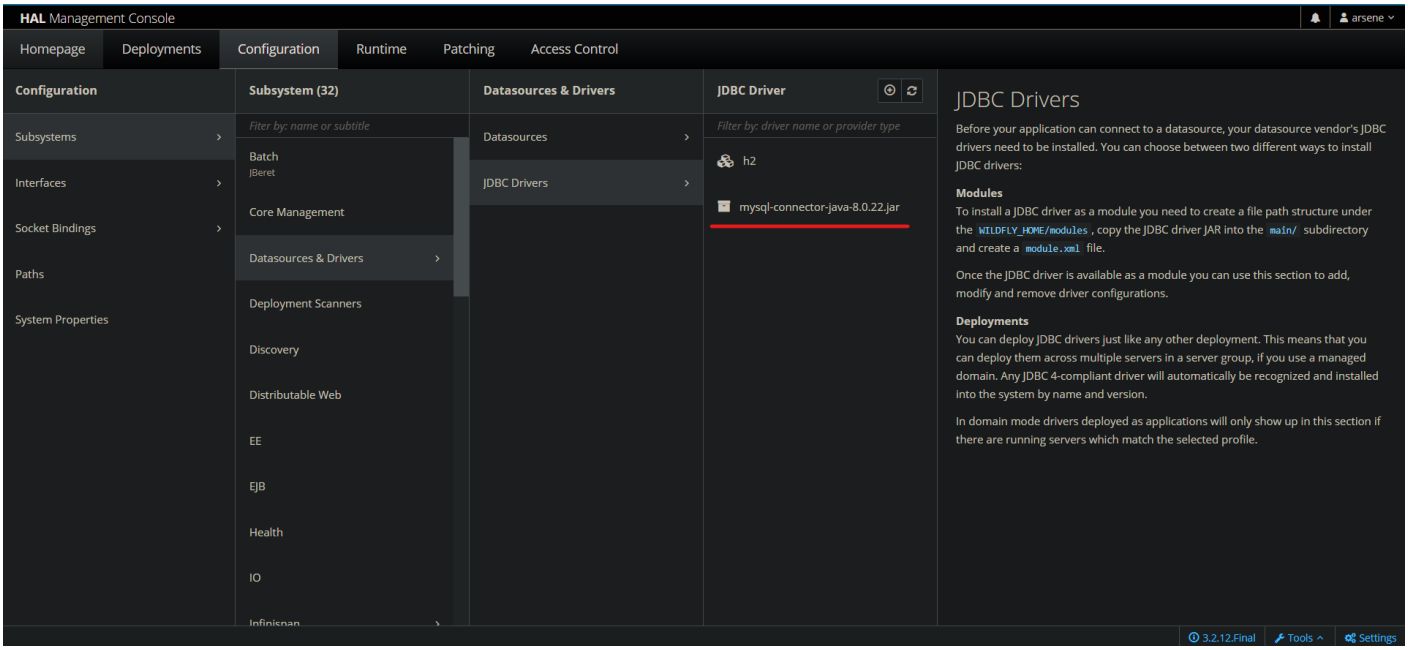
Ouvrez maintenant l'interface d'administration à `http://localhost:9990` et rendez vous dans l'onglet *Deployement*. Ajoutez ensuite un nouveau déploiement en cliquant sur le bouton + puis *Upload Deployment*:



Dans la fenêtre qui s'ouvre, uploadez votre JAR JDBC et validez. Si tout ce passe bien il devrait être répertorié ici dans les déploiements :



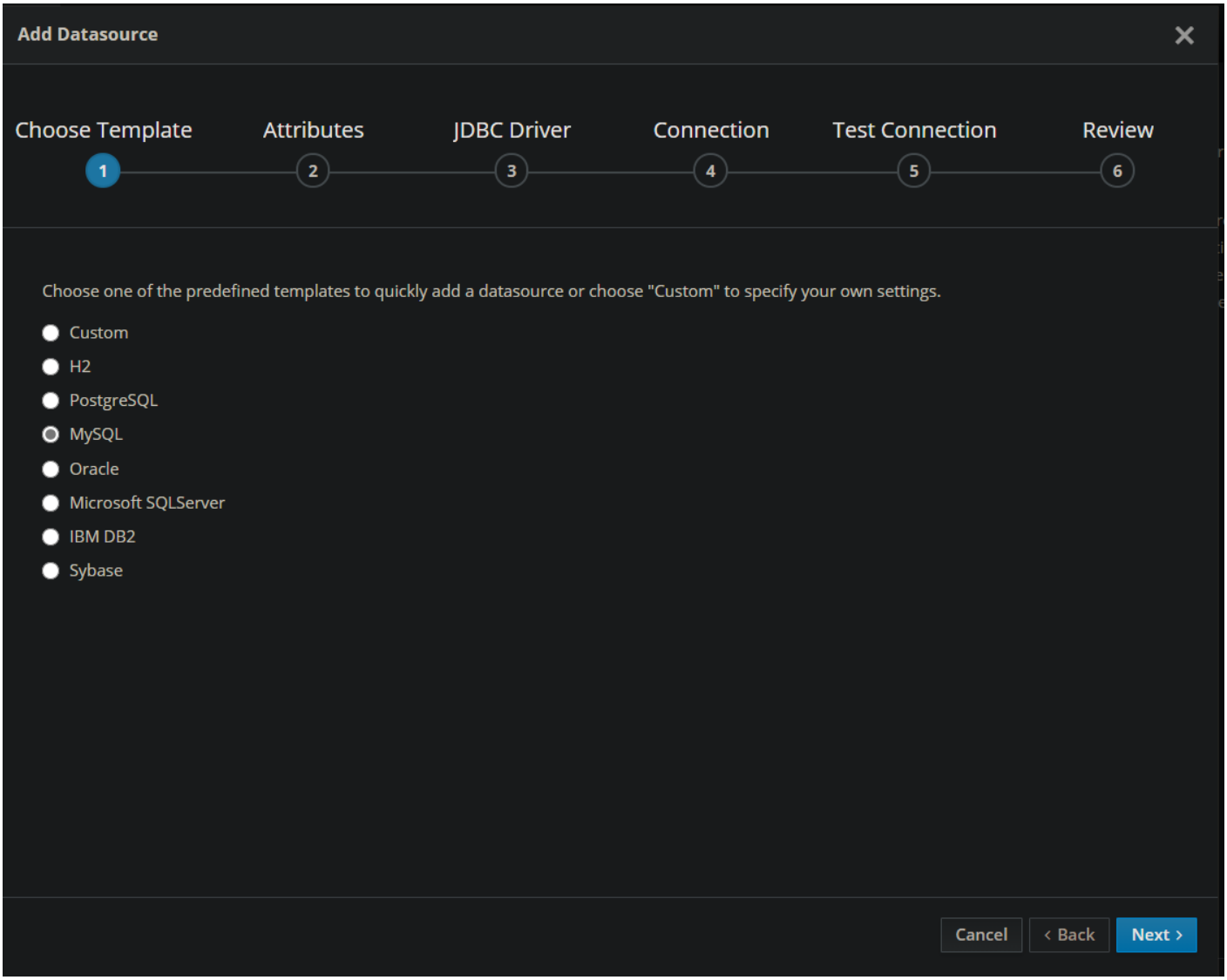
Ainsi qu'ici dans l'onglet *Configuration* sous *Configuration -> Subsystems -> Datasources & Drivers -> JDBC Drivers*.



Configurez la source de donnée

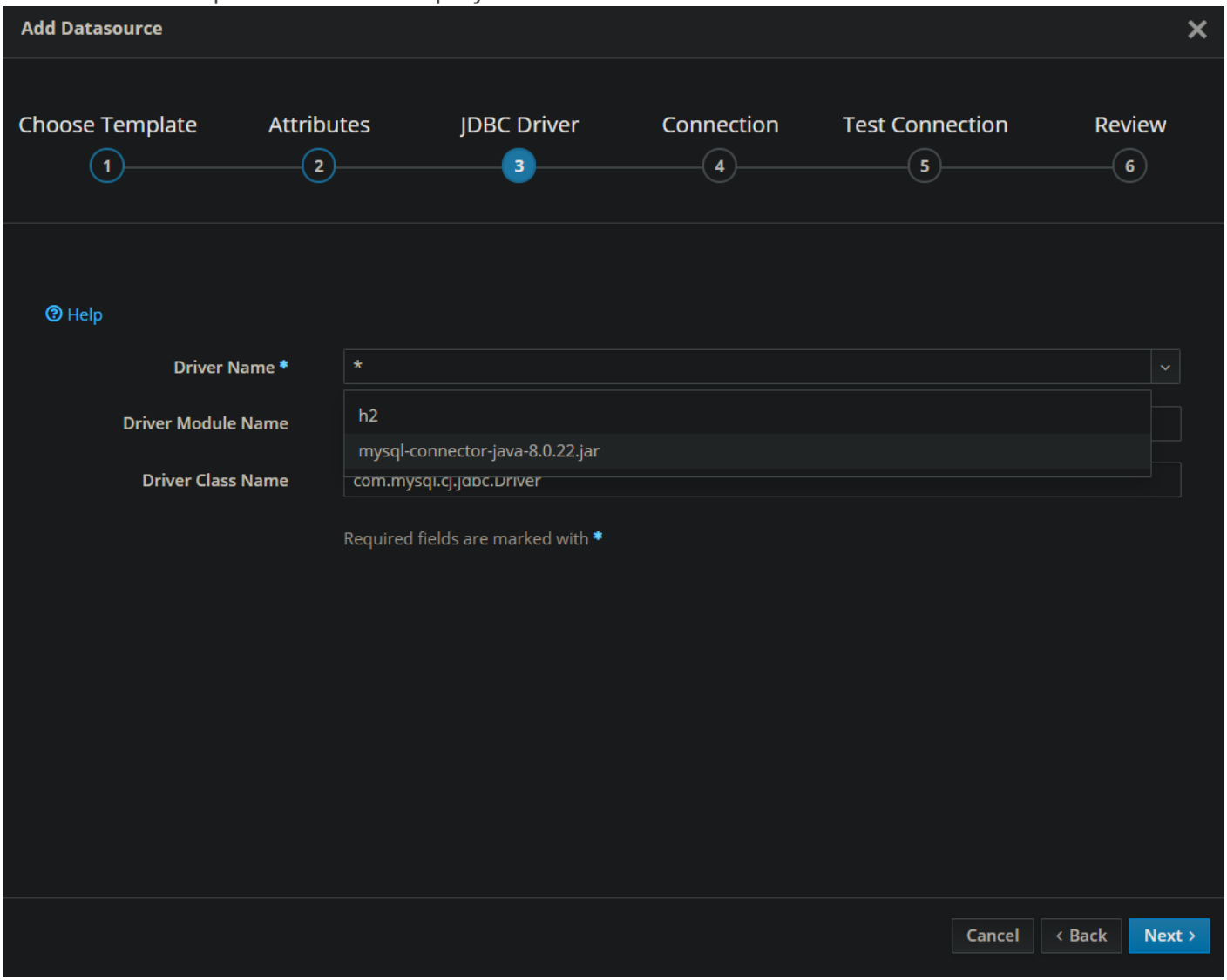
Sur l'interface d'administration de Wildfly, allez à *Configuration -> Subsystems -> Datasources & Drivers -> Datasources* et utilisez le boutons + et New Datasource*. Suivez ensuite les instructions du wizard en renseignant les informations.

Choisir notre SGBD :



Dans attributes, vous pouvez choisir un nom pour votre Datasource ainsi que son adresse dans l'annuaire JNDI.

Choisir le driver que nous avons déployé :



Pour les identifiants de votre base de donnée voici comment les renseigner :

- URL : `jdbc:mysql://localhost:3306/<nom de la base>`
- Username : votre login mysql
- Password : votre mot de passe mysql

Vous pouvez utiliser le bouton Test Connection pour vérifier que les information son correctes.

> Il m'est déjà arrivé que le bouton Test Connection ne marche pas mais que la datasource soit tout de même bien configurée et fonctionnelle

> Si au moment du déploiment de votre artéfact sur le serveur MySQL vous lance une erreur de TimeZone, ajoutez ceci à la fin de l'URL JDBC : `?useLegacyDatetimeCode=false?useUnicode=true&useJDBCCompliantTimezoneShift=true&`

> `useLegacyDatetimeCode=false&serverTimezone=UTC`. Vous pouvez modifier votre datasource existante en allant dans *Configuration -> Subsystems -> Datasources & Drivers -> Datasources* Et en faisant "View" sur votre datasource, puis onglet "Connection" puis "Edit"

Utiliser le nouvelle source de donnée

Dans votre projet, sous le dossier `resources`, créez un dossier, `META-INF` et dans ce dernier un fichier, `persistence.xml`, et entrer ce contenu pour référencer votre datasource dans votre projet :

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd"
             version="2.2">
    <persistence-unit name="default">
        <jta-data-source>java:/MySqlDS</jta-data-source>
        <properties>
            <property name="hibernate.hbm2ddl.auto" value="update"/>
            <property name="hibernate.dialect" value="org.hibernate.dialect.MariaDBDialect"/>
        </properties>
    </persistence-unit>
</persistence>
```

Voilà, votre datasource est configurée, nous allons pouvoir maintenant étudier plus en détail les outils proposés par JPA.

EJB Entity

Les EJB Entity sont des EJB particuliers qui servent à représenter des données. Pour créer un EJB entité, il faut créer une classe annotée avec l'annotation `@Entity`. Une classe entité doit avoir un attribut annoté avec `@Id` qui correspondra à la clé primaire dans la table de la base de donnée. Chaque propriété de l'entité sera mappée à la colonne de la base de donnée. Les autres champs de la classe entités sont persistés automatiquement. Pour qu'un champs ne soit pas persisté, il faut l'annoter avec `@Transient`.

Exemple d'entité :

```
@Entity
```

```
public class Todo {
  @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;

  private String title;
    private String description;

  ... constructeur & getters & setters ...
}
```

Ici l'annotation `@GeneratedValue(strategy= GenerationType.AUTO)` permet de générer automatiquement l'id avec un auto-incrément.

On peut aussi définir des contraintes sur les colonnes en rajoutant l'annotation `@Column` sur le champs et en lui passant des paramètres, par exemple :

```
@Column(unique = true, length = 32)
```

Cela permet de mettre la contraint "unique" sur le champs et d'imposer une longueur maximum de 32 caractère.

Entity Manager

L'entity manager est une interface qui permet d'intéragir avec les entités :

- Les persister
- Les supprimer
- Faire des requêtes

Pour l'obtenir dans une classe, demandez le avec l'annotation `@PersistenceContext` :

```
public class TodoRepository {
    @PersistenceContext
    private EntityManager entityManager;
}
```

L'entity manager vas vous permettre de faire toutes les opérations nécessaires sur les Entity.

Persister une Entity

Utilisez la méthode `persist()` :

```
entityManager.persist(new Todo("test title", "test content"));
```

Supprimer une Entity

Utilisez la méthode `remove()` :

```
entityManager.remove(myEntity);
```

Récupérer une Entity

Pour récupérer une entité à partir de son Id :

```
Todo todo = entityManager.find(Todo.class, todoId);
```

Créer une Requête JPQL

JPQL pour Java Persistence Query Language est un langage de requête orienté objet qui permet de créer des requêtes sur des entités. Par exemple :

```
List<Todo> todos = entityManager.createQuery("SELECT t FROM Todo t",
Todo.class).getResultList();
```

Ou alors une requête paramètrée :

```
String title = "test title";
List<Todo> todos = entityManager.createQuery("SELECT t FROM Todo t WHERE t.title = :title",
Todo.class)
  .setParameter("title", title)
  .getResultList();
```

Relations

Les relations permettent de faire des liens entre les entités. Il existe quatres types de relations, définies par des annotations :

- `@ManyToOne` : plusieurs instances de cette classe entité sont en relation avec une unique instance d'une autre (ex: Pages d'un livre - plusieurs pages sont reliées à un unique livre)
- `OneToMany` : une unique instance de cette classe entité est en relations avec plusieurs instances d'une autre (ex: Livre qui contient des pages - un unique livre est relié à plusieurs pages)
- `@ManyToMany` : plusieurs instance de cette classe entité sont en relations avec plusieurs isntance d'une autre (ex: Classes et Professeurs - Les professeurs ont plusieurs classes et

les classes ont plusieurs professeurs.
- `@OneToOne` : une unique instance de cette classe entités est reliée à une unique instance d'une autre (ex: Dircteur et Ecole - une directeur dirige une seule école et une école est dirrigée par un seul directeur).

Exemple :

```
@Entity
public class Todo {
  ... autres propriétés ...

  @ManyToOne
  private ApplicationUser owner;

  ... getters & setters ....
}
```

```
@Entity
public class ApplicationUser {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private String id;

  private String name;

  @OneToMany(
    cascade = CascadeType.ALL,
        orphanRemoval = true
    )
    private Set<Todo> todos;

    ... getters & setters ...
}
```

Les annotations de relation possèdes plusieurs paramètres utiles à connaitres :

- `fetch` :
  - `FetchType.EAGER` : charge les données de la relation directement (défaut pour `@ManyToOne` et `@OneToOne`)
  - `FtechType.LAZY` : charge les données de la relation que quand le getter est appelé (défaut pour `@OneToMany` et `@ManyToMany`)

- `orphanRemoval` : `true` ou `false`, détermine dans un `@OneToMany` si les enfant doivent être supprimés quand le parent est supprimé → un enfant ne peut exister sans parent.
- `cascade` : permet de dire quelles opérations de persistance sont propagées du parent vers l'enfant

WebServices JAX-RS

g