JPA

JEE et EJB

Rôle de JEE

- Utilisaton de concepts validés
- Consorsium d'entreprise concernés
- Normalisation et interopérabilité
- Mise en valeur des spécificité d'un *Vendor*
    - notamment en cluster

Différence entre Spring et JEE

- Spring est novateur
- Spring est paradoxalement plus stable
- JEE n'a pas pour objectif d'innover
- Spring fonctionne dans Tomcat ou en Standalone
- Il y a une galaxie de projets Spring
- Parties communes entre Spring et JEE (Servlet, Hibernate)

Principales parties de Spring/JEE

- L'injection de Dépendence (Spring core / CDI)
- La servlet et JAX-RS
- La vue : JSP & JSF
- Le Service/EJB
    - Mecanisme de proxy
- Hibernate (JPA) ou DAO

Rôles des EJB

- Outil phare de JEE
- Mais une partie seulement des specs de JEE

Pourquoi les EJB

Se concentrer sur le code métier, et non pas sur la plomberie :

- Qui crée l'objet ?
- Multiplication des constructeurs
- Manipulation - donc source d'erreur - de la sécurité
- etc.

En résumé :

- EJB est un model de *composant serveur standard* pour des applications business distribuées
- EJB Résoudre des problèmes communs avec des technologies standards et validées
- EJB est plus productif

Injection de dépendance


```
public class FlagBusiness {

    @EJB ForaDataSource datasource;
    @Inject ForaLogger logger;

    public void flagComment(String commentId, String flagMessage){
        logger.log("flagging comment "+commentId+" with "+flagMessage);
    }
}
```

- EJB : Entreprise JavaBean : *gros* composant
- CDI : Context and Dependency Injection
- CDI va se retrouver dans les autres parties
- Peu de risque de NullPointerException
- Pas de multiple constructeurs à maintenir
- Plus productif, plus lisible

```
@Path("/comment")
public class MessageRestService {

    @Inject CommentService service;

    @GET
    @Path("/{id}")
    public Response printMessage(@PathParam("id") long id) {

        String result = service.getComment(id);
        return Response.status(200).entity(result).build();
    }
}
```

- JAX-RS crée une servlet
- Une Servlet point d'entrée de l'application

ForaDataSource.java :

```
@Singleton
@Named("fora")
public class ForaDataSource {
    public List<Comment> getComments(){...}
}
```

index.xhtml :

```
<ui:repeat var="comment" value="#{fora.comments}">
<tr>
    <td>#{comment.content}</td>
    <td>#{comment.id}</td>
    <td>#{o.price}</td>
    <td>#{o.qty}</td>
</tr>
</ui:repeat>
```

- Validation des données avant stockage
- Relation entre tables
- Modèle objet (Domain driven design)

```
@Entity
public class Subject{

    @Id
    @GeneratedValue
    long id;

    @Size(min=4, max=30)
    String title;
    String content;

    @OneToMany(targetEntity = Comment.class)
    List<Comment> comments = new ArrayList<Comment>();

    @ManyToOne(fetch = FetchType.EAGER)
    User user;
}
```

- BeanValidation : Validation != Exception
- Mapping vers la base

```
@Stateless
public class SubjectBusiness {

    @PersistenceContext
    EntityManager em;

    public Comment getSubjectById(String id){
        return em.find(Comment.class, id);
    }

    public Subject getEagerSubjectById(long id) {

        String q = "Select s from Subject s JOIN FETCH s.comments WHERE s.id = :id";
        TypedQuery<Subject> tQuery = em.createQuery(q,Subject.class);
        tQuery.setParameter("id", id);

        return tQuery.getSingleResult();
    }
```

Les avantages

- Génération de Requête
- Gestion des transactions
- Mise en Cache
- On travail directement avec des objets **typés**

```
@Stateless
@SecurityDomain("MyRealm")
public class CitationEJB {

    @Resource SessionContext sessionContext;

    @RolesAllowed("user") //standard user
    public String getUserCitation(){
        String user = sessionContext.getCallerPrincipal().getName();
        return citation + " from "+user;
    }
}
```

Autres usages des EJB

- EJB "message-driven"
- Intercepteurs
- Timers
- JMX Monitoring

Hibernate et JPA

Installation et Connexion

- serveur : Création d'une datasource
- projet : Relation entre persistence.xml et datasource

Persistence.xml

Création du fichier

- Créer src/META-INF/persistence.xml
- Vérifier qu'il apparait dans WEB-INF/classes/META-INF/persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="...">
   <persistence-unit name="persistenceUnit" transaction-type="JTA">
     <provider>org.hibernate.ejb.HibernatePersistence</provider>
     <jta-data-source>java:jboss/datasources/fora</jta-data-source>
     <properties>
       <property name="hibernate.dialect"
            value="org.hibernate.dialect.MySQLDialect" />
       <property name="hibernate.hbm2ddl.auto" value="create-drop" />
       <property name="hibernate.show_sql" value="false" />
     </properties>
  </persistence-unit>
</persistence>
```

Création d'une Entity


```
@Entity
public class User implements Serializable{


    private static final long serialVersionUID = 34903731994788167 86L;

    @Id
    @GeneratedValue
    Long id;
    String email;
    String firstname;
    String lastname;

    int version=1;
}
```

Test de la persistence


```
@Singleton
@Startup
public class TestDatabase {

    @PersistenceContext EntityManager em;
    @EJB ForaDataSource fora;

    @PostConstruct
    @TransactionAttribute
    public void fillDb() {
        User nicolas = new User();
        nicolas.setEmail("nz@robusta.io");
        nicolas.setFirstname("Nicolas");
        em.persist(nicolas);
        em.flush();//don't stay in cache
    }
}
```

Principes de base

- Mapping par annotation (et implicite)
- EntityManager
- DAO simplifié
- JPQL

CRUD

JPA permet de :

- Create : `em.persist(entity)`
- Read : `em.find(Class, id)`
- Update : transparent : `entity.setName("john")`
- Delete : `em.remove(entity)`

TP : Faire la couche CRUD **simplissime** de User et Comment

Principes mécaniques

- persistence.xml
- proxy
- Thread et Transaction
- collections
- lazy/eager

Mapping

Annotations fréquentes

Définition des entity

- `@Entity`
- `@Id` : Un `Id` obligatoire
- `@GeneratedValue(strategy = GenerationType.AUTO)`
- `@Basic` : optionel, permet des attributs
    - `@Basic(fetch=FetchType.LAZY)`
- `@Transient`

Mapping des noms

- `@Table`
- `@Column`
- `@JoinColumns`, `@JoinColumn`

Collections et multiway

- `@OneToMany` : listes classiques
- `@OneToOne` : facultatif
- `@ManyToMany`: ex : favoriteComments
- `@ManyToOne` : Bidirectional relation

- Faire le Mapping de :
    - User
    - Account
    - Topic
    - Comment
- Utiliser `@Transient` en cas de problème

Cas plus spéciaux

- `@Embedded`
- `@Lob`
- `@Enumerated`
- `@Temporal`
- `@Version`
- `@OrderBy`
- `@IdClass`, `@Embeddable`, `@EmbeddedId`

Exemple


```
@Entity
public class Comment {

   @Temporal(TemporalType.TIMESTAMP)
   Date date;
}

@Entity
public class Topic {

   @OneToMany
   @OrderBy("date ASC")
   List<Comment> comments = new ArrayList<Comment>();
}
```

TP : Mapping évolué

- Définir les *favorite comments* d'un User
- Rajouter la relation *topic* d'un Comment
- Supprimer un topic : qu'en est-il des comments ?

JPQL

Langage

- `List getResultList();`
- `Object getSingleResult();`
- `int executeUpdate();`
- `Query setParameter( String name, Object value);`
- `Query setParameter( int position, Object value);`

Exemples


```
Query q1 = em.createQuery("SELECT u FROM User u WHERE u.money > :money");
q1.setParameter("money", 12);
List users = q1.getResultList();

TypedQuery<Country> q2 =
    em.createQuery("SELECT u FROM User u WHERE u.money > :money", User.class);
q1.setParameter("money", 12);
List<User> users = q1.getResultList();
```

Jointures

Il existe deux méthodes de jointures

- Traverse d'objets
- Utilisation de `JOIN` : lazy
- utilisation de `JOIN FETCH` : eager

Cette requête ramène tous les commentaires des *admins* :

```
String jpql = "SELECT c From Comment c WHERE c.user.account.role = 'admin'";


TypedQuery<Comment> tq = em.createQuery(jpql, Comment.class);
List<Comment> adminComments = tq.getResultList();
```

- utilisée pour *sélectionner* les données :
- Productif
- Facilite la lecture du code

JOIN

La jointure par le mot clé `JOIN` est utilisée pour relier des données

```
String jpql = "SELECT c From Comment c JOIN c.warnings";
```

Comme le JOIN en SQL, cette requête ne selectionne que les Commentaires ayant des warnings. Par contre ici les warnings ne sont pas remontés de la base.

```
TypedQuery<Comment> tq = em.createQuery(jpql, Comment.class);
List<Comment> warningComments = tq.getResultList();

// Déclenche une requête LAZY !
String message = warningComments.get(0).getContent();
```

JOIN FETCH

`JOIN FETCH` est utilisé pour remonter des grapes de données

```
String jpql = "SELECT c From Comment c JOIN FETCH c.user";

TypedQuery<Comment> tq = em.createQuery(jpql, Comment.class);
List<Comment> notAnonymousComments = tq.getResultList();

// Ne déclenche pas de requête !
User u0 = notAnonymousComments.get(0).getUser();
```

Dans la litterature

Attention à :

- Ne pas remonter toute la base en une fois
- Inversement, ne pas faire 150 requêtes
    - On parle du problème ** n+1 select **
    - Very very famous !