

# Framework Java EE

1

## De quoi parle t'on ?

- JavaEE (version 6 & 7)
- Architecture d'entreprise
- Services web
- Développement

2

## Le plan !

- Java EE, definition
- Les applications d'entreprise
- Entités
- Persistance (JPA)
- Entreprise Java Bean
- Java Server Faces
- Binding XML
- Webservice SOAP
- WebService REST

3

## Les outils du cours

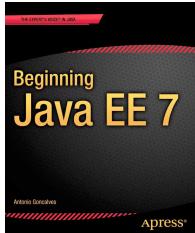


- Docker
- Java 8, Maven, Git **maven**
- IDE : IntelliJ
- Serveur d'application : Wildfly
- Base de donnée : Postgres
- pgAdmin 4

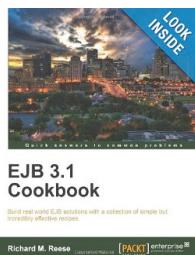


4

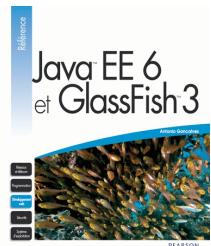
## Références



Beginning Java EE 7  
Antonio Goncalves  
aPress 2013



EJB 3.1  
Cookbook  
Richard M. Reese  
Packt Publishing 2011



Java EE 6 & Glassfish 3  
Antonio Goncalves  
Pearson 2010

5

*Découvrons les outils*

7

## Règles d'or

*N'hésitez pas à interrompre*

*Il n'y a pas de questions stupides*

*JavaEE est complexe, même moi je ne comprends pas tout*

*Parlez entre vous, partagez*

*Et surtout n'hésitez pas à interrompre*

6

## Java EE 7

**Un standard**

C'est un ensemble de plus de 30 spécifications

Coiffées par la JSR 342

**Et réparties dans 5 domaines**

Web Application Technologies   Enterprise Application Technologies   Web Services Technologies

Management and Security Technologies   Java EE-related Specs in Java SE

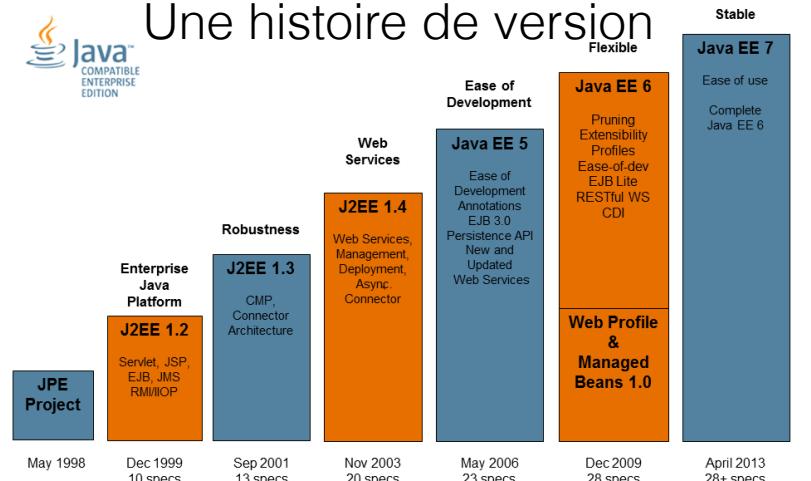
<http://www.oracle.com/technetwork/java/javase/tech/index.html>

8



# JavaEE 7

## Une histoire de version

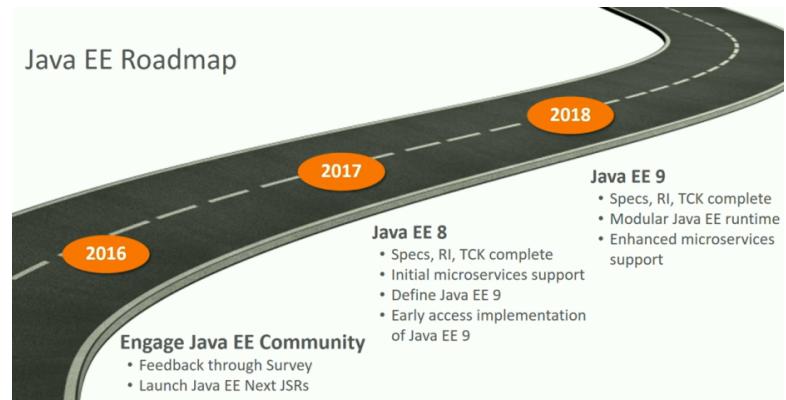


9

# JavaEE 8

*Et le futur ?*

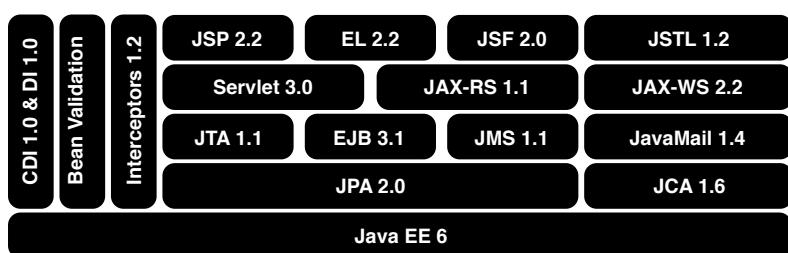
## Java EE Roadmap



10

# JavaEE 6

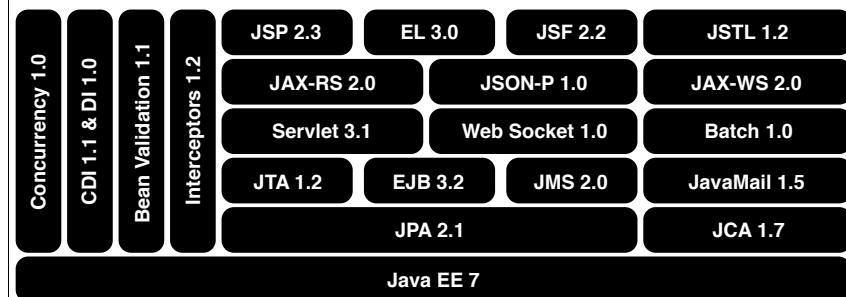
## Les principaux composants



11

# JavaEE 7

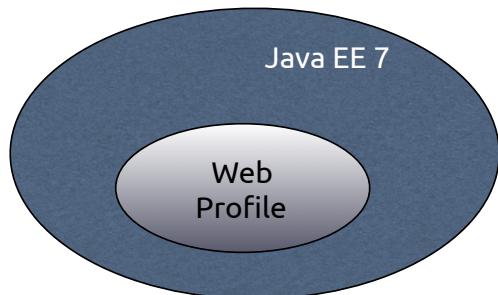
## Les principaux composants



12

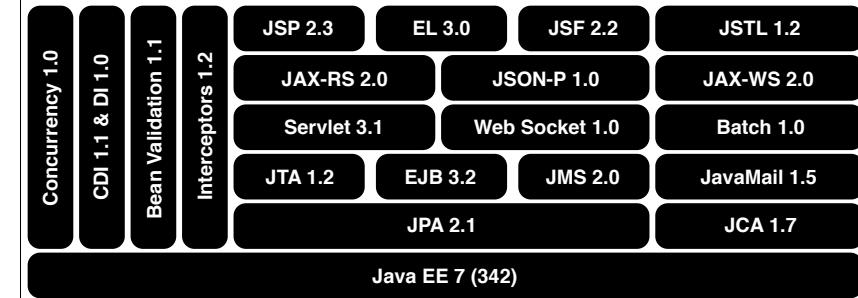
## Java EE profile web

Version légère



13

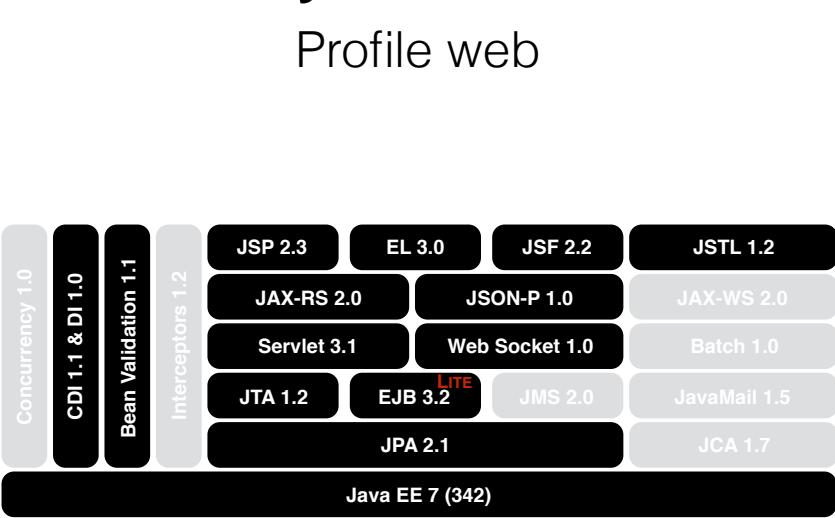
## JavaEE 7



14

## JavaEE 7

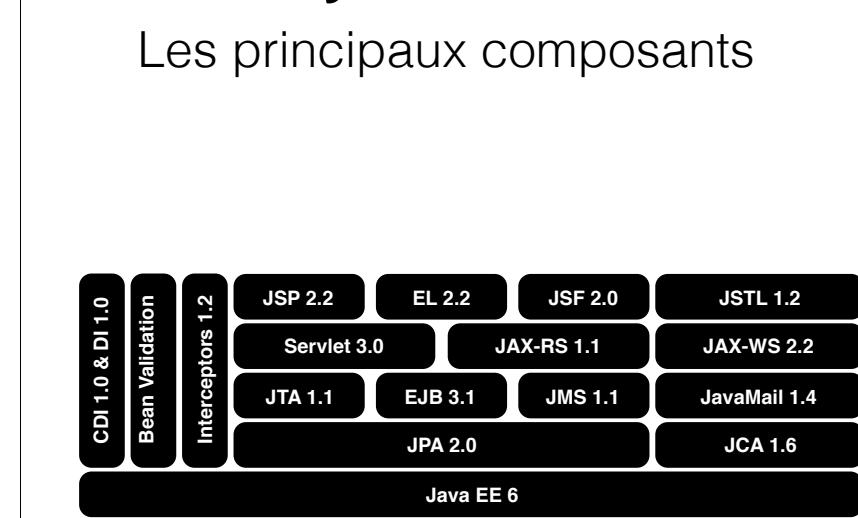
Profile web



15

## JavaEE 6

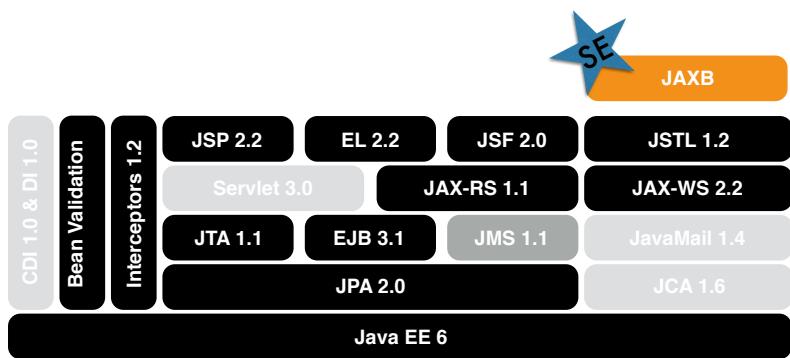
Les principaux composants



16

# JavaEE 6

Les principaux composants  
Abordé en formation



17

# Pourquoi JavaEE 6 & 7



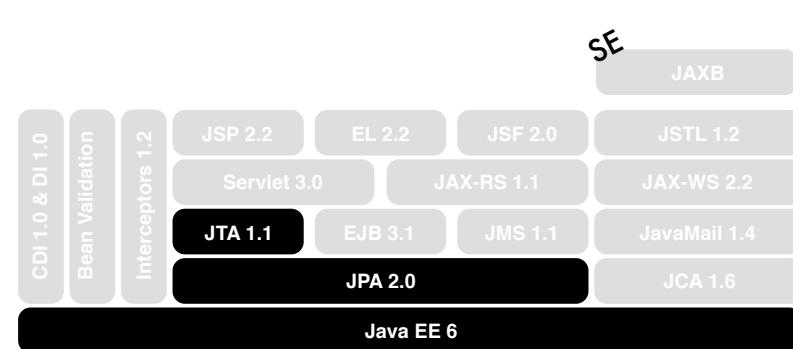
18

# JPA

Les entités et JPA



19



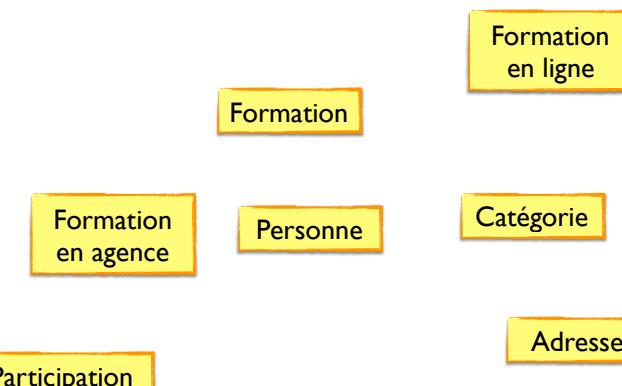
20

# Modèle du domaine

- Image conceptuelle du problème que l'application essaie de résoudre
- Ensemble d'objets (physique ou concept) et de relations ou associations entre ces objets

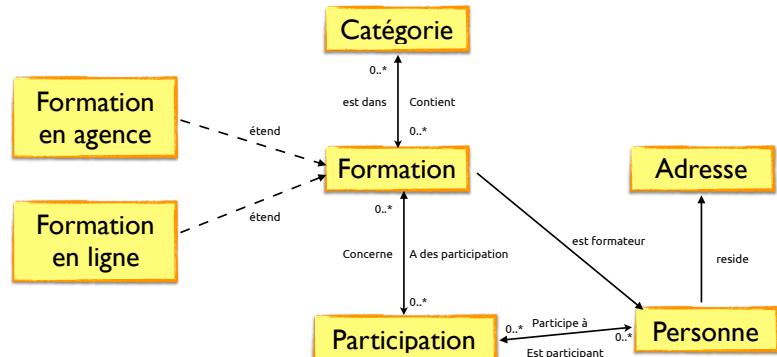
21

# Exemple



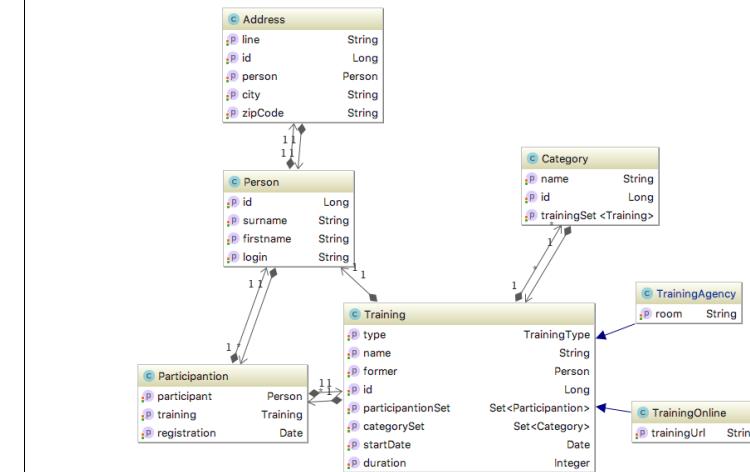
22

# Exemple



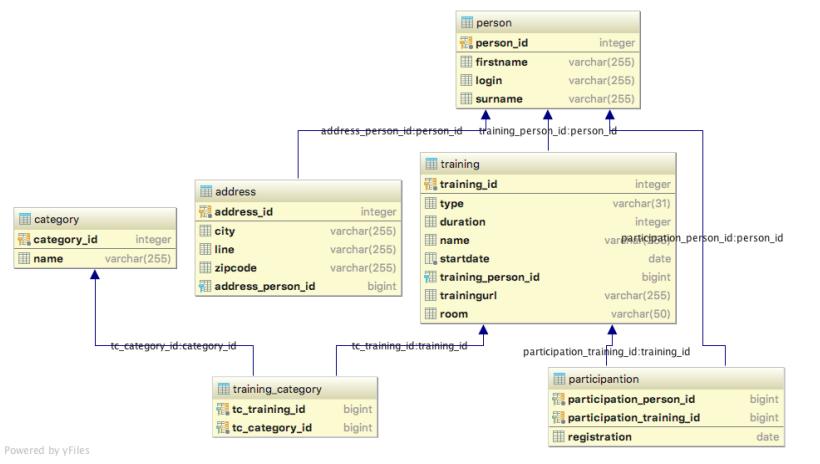
23

# Modele UML



24

# Modele Relationnel



25

# Modèle riche ou anémique

- un modèle anémique est un modèle qui n'encapsule que les données en se contentant de mapper directement les classes sur les tables dans une relation un pour un.
- un modèle riche est un modèle qui encapsule les données mais aussi le comportement en utilisant des notions comme l'héritage, le polymorphisme et l'encapsulation
- Un modèle anémique est plus simple à réaliser, mais un modèle riche est plus pratique pour la programmation de l'application

27

# Acteurs du modèle

- Objets : traduit par les classes java dont les attributs seront les données (Personne contient nom, prénom,...)
- Relations : traduit par des classes ayant des références sur d'autre. Ces relations peuvent être unidirectionnelles ou bidirectionnelles
- Multiplicité ou cardinalité : les relations ne sont pas forcément un pour un , on trouve donc one-to-one, one-to-many, many-to-one et many-to-many

26

# Objet du domaine

```
public class Person {  
    /**  
     * Identifiant  
     */  
    private Long id;  
  
    /**  
     * Login  
     */  
    private String login;  
  
    /**  
     * Nom de famille  
     */  
    private String surname;  
  
    /**  
     * Prénom  
     */  
    private String firstname;  
}
```

28

## Anatomie d'une entité

- Simple POJO
- Serializable
- Annoté avec `@Entity`
- Possède un clé primaire (simple ou non)
- Constructeur sans arguments
- Classe non finale

29

## Entité ou Pojo ?

- Une entité est un POJO
- Mais un POJO n'est pas une entité
- Les entités sont managées par EntityManager
  - Persiste les entités
  - Leur état est synchronisé avec la base
- Quand ils ne sont pas managés, ce sont de simple POJO

31

## Entité

```
@Entity
public class Person implements Serializable {

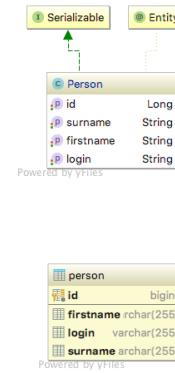
    /**
     * Identifiant
     */
    @Id
    private Long id;

    /**
     * Login
     */
    private String login;

    /**
     * Nom de famille
     */
    private String surname;

    /**
     * Prénom
     */
    private String firstname;

}
```



30

## Les annotations

- Les annotations peuvent être mises sur l'attribut lui-même (Field-based persistence) ou sur le getter de l'attribut (Property-based persistence)
- Le choix exclusif
- l'utilisation du property-based respecte mieux l'encapsulation pronée en POO

32

# Type de donnée

Types	Exemple
Primitives	int, double, long
Wrapper de primitive	Integer, Double, Long
Chaine	String
Type serialisable	BigInteger, java.sql.Date
Classe serialisable (utilisateur)	Implementant java.io.Serializable
Tableau	Byte[]
Type enuméré	Tous les énumérés
Collection (entité)	Set
Type rattaché	classes annotées @Embeddable

33

# Les annotations de mapping

- `@Table`
- `@Column`
- `@Enumerated`
- `@Lob`
- `@Temporal`
- `@Embeddable`
- `@Transient`

34

## `@Table`

```
@Target({TYPE}) @Retention(RUNTIME)
public @interface Table {
    String name() default AB;
    String catalog() default AB;
    String schema() default AB;
    UniqueConstraint[] uniqueConstraints()
default {};
}
```

35

## `@Table`

- **Optionnelle**, si elle est omise la table a le nom de la classe, dans le schéma par défaut
- **name** : permet de préciser le nom de la table
- **schema** : permet de préciser le schéma dans la base
- **catalog** : permet de préciser le catalogue dans la base
- **uniqueConstraints** permet de préciser une contrainte d'unicité sur une ou des colonnes en cas d'utilisation de la génération de schéma

36

## @Column

```
Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Column {
    String name() default FG;
    boolean unique() default false;
    boolean nullable() default true;
    boolean insertable() default true;
    boolean updatable() default true;
    String columnDefinition() default FG;
    String table() default FG;
    int length() default 255;
    int precision() default 0;
    int scale() default 0;
}
```

37

## @Column

- **optionnelle**, si omise, la colonne à le nom de l'attribut ou propriété
- **name** : nom de la colonne
- **table** : en cas de mapping sur deux tables
- **insertable, updatable** : permet d'exclure le champs des requêtes INSERT ou UPDATE, utiliser pour les champs en lecture seul comme les clés primaires générées
- Les autres sont principalement pour la création de table :
  - **nullable** : si la colonne supporte les valeurs nulles
  - **unique** : si la colonne à une contrainte d'unicité
  - **length** : taille de la colonne
  - **precision, scale** : pour les décimales
  - **columnDefinition** : Spécifie le SQL exact de la création

38

## @Enumerated

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Enumerated {
    EnumType value() default ORDINAL;
}
public enum EnumType{
    ORDINAL,
    STRING }
```

39

## @Enumerated

- Si on utilise ORDINAL, les valeurs sauvegardées en base sont de 0 à n selon le nombre d'éléments dans l'énumération
- Si c'est STRING, c'est le nom de la valeur de l'énuméré qui est stocké
- Si l'annotation est omise, l'attribut ou propriété est sauvegardée de façon ordinal

40

## @Lob

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Lob {
}
```

41

## @Lob

- permet de désigner un champs comme BLOB ou CLOB
- Si le champs est un char[] ou String c'est un CLOB, sinon c'est BLOB
- Généralement utilisé en conjonction avec @Basic (fetch=FetchType.LAZY) permettant d'en différer le chargement

42

## @Temporal

```
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Temporal {
    TemporalType value();
}
public enum TemporalType{
    DATE,
    TIME,
    TIMESTAMP
}
```

43

## @Temporal

- Cette annotation est redondante avec les types java.sql.Date, java.sql.Time, java.sql.Timestamp
- Permet de préciser si on veut persister les types java.util.Date et java.util.Calendar sur des champs de type DATE, TIME ou TIMESTAMP
- Sans annotation le persistence provider utilise le type TIMESTAMP

44

## @Transient

```
@Target({ElementType.METHOD, ElementType.FIELD})  
@Retention(RetentionPolicy.RUNTIME)  
public @interface Transient {  
}
```

45

## @Transient

- Permet de ne pas persister une donnée qui pourrait être calculée

```
@Entity  
@Table(name = "PERSON")  
public class Person {  
    @Id  
    @GeneratedValue(generator = "USER_SEQ")  
    private Long id;  
    @Column(nullable = false)  
    private String name;  
    @Temporal(TemporalType.DATE)  
    private Date dateOfBirth;  
    @Transient  
    private int age;  
  
    public Person() {  
    }  
}
```

46

## @SecondaryTables

Il est parfois nécessaire de stocker les valeurs dans différentes tables

```
@Target({TYPE})  
@Retention(RUNTIME)  
public @interface SecondaryTables {  
    javax.persistence.SecondaryTable[] value();  
}
```

47

## @ SecondaryTables

```
@Entity  
@Table(name = "PERSON")  
@SecondaryTables({  
    @SecondaryTable(name = "CITY"),  
    @SecondaryTable(name = "COUNTRY")  
})  
public class Person {  
    @Id  
    @GeneratedValue(generator = "USER_SEQ")  
    private Long id;  
    @Column(nullable = false)  
    private String name;  
    private String Street1;  
    private String Street2;  
    @Column(table = "CITY")  
    private String city;  
    @Column(table = "COUNTRY")  
    private String country;
```

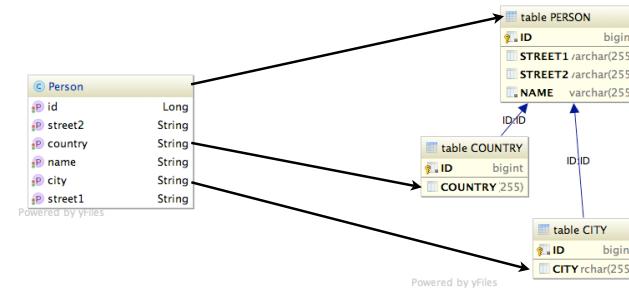
48

## @SecondaryTables

```
@Entity  
@Table(name = "PERSON")  
@SecondaryTables({  
    @SecondaryTable(name =  
    "CITY", pkJoinColumns=@PrimaryKeyJoinColumn(name= "ID")),  
    @SecondaryTable(name =  
    "COUNTRY", pkJoinColumns=@PrimaryKeyJoinColumn(name= "ID"))  
})  
public class Person {  
    @Id  
    @GeneratedValue(generator = "USER_SEQ")  
    private Long id;  
    @Column(nullable = false)  
    private String name;  
    private String Street1;  
    private String Street2;  
    @Column(table = "CITY")  
    private String city;  
    @Column(table = "COUNTRY")  
    private String country;  
}
```

49

## @ SecondaryTables



50

## @ SecondaryTables

- Permet d'associer des tables secondaires
- soit une, utilisation directe de `@SecondaryTable`
- soit plusieurs (`@SecondaryTables & @SecondaryTable`)
- Risque non négligeable sur les performances

51

## Les classes «embarquées» Embedded Class

- L'inverse des tables secondaires
- Entité et embeddedClass == 1 seule table
- La classe embarquée n'est pas une entité
- Annoté `@Embeddable`
- N'a pas d'`@Id`
- Utilisable dans plusieurs entités

52

## EmbeddedClass

### Exemple

```
@Entity
@Table(name = "PERSON")
public class Person {
    @Id
    @GeneratedValue(generator = "USER_SEQ")
    private Long id;
    @Column(nullable = false)
    private String name;
    @Embedded
    private Address address;

    public Person() {
    }
}
```

53

## EmbeddedClass

### Exemple - réutilisation

```
@Entity
@Table(name = "CLUB")
public class Club {

    @Id
    private long id;
    @Column(nullable = false, length = 200)
    private String name;
    @Column(nullable = true)
    private String motto;
    @Embedded
    private Address address;

    public Club() {
    }
}
```

55

## EmbeddedClass

### Exemple

```
@Embeddable
public class Address {

    private String street1;
    private String street2;
    private String zipCode;
    private String city;
    private String country;

    public Address() {
    }
    @Column(length=50, nullable=false)
    public getStreet1() {
        return this.street1;
    }
}
```

54

## Identifiants

- L'identifiant d'une entité est la clé primaire
- Ce peut être un champs de type primitif, un ensemble de champs ou un objet
- Il existe 3 moyens de le définir
  - `@Id`
  - `@EmbeddedId`
  - `@IdClass`

56

## @Id

- Permet de marquer un champs comme étant l'identifiant de l'objet
- Peut être un type primitif, un wrapper de primitif ou un Serializable (java.lang.String, java.util.Date, java.sql.Date)
- Il est recommandé d'éviter les types float, double et leur wrapper, à cause de la précision de la base
- De même il faut éviter le type TimeStamp

57

## @EmbeddedId

- Utilisation d'un objet embarqué comme Identifiant
- **La classe doit redéfinir hashCode et equals**
- Permet une maintenance du code simple
- Modifie le modèle du domaine

58

## @EmbeddedId

```
@Entity
public class Category {
    @EmbeddedId
    private CategoryPK categoryPK;

    public Category() {
    }
    ...
}

@Embeddable
public class CategoryPK implements Serializable {
    private String Name;
    private Date createDate;

    public CategoryPK() {}
    @Override public boolean equals(Object o) {}
    @Override public int hashCode() {}
}
```

59

## @IdClass

- Permet d'utiliser plus d'une annotation @Id dans une Entité
- Nécessite la définition d'une classe supplémentaire qui implémente java.io.Serializable et fournit des implémentations correctes de **equals et hashCode**
- Peut poser des problèmes de maintenance mais maintient l'état du modèle du domaine

60

## @IdClass

```
@Entity  
@IdClass(CategoryPK.class)  
public class Category implements Serializable {  
    @Id  
    private String Name;  
    @Id  
    private Date createDate;  
  
    public Category() {}  
    ...  
}  
  
public class CategoryPK implements Serializable {  
    private String Name;  
    private Date createDate;  
  
    public CategoryPK() {}  
    @Override public boolean equals(Object o) {}  
    @Override public int hashCode() {}  
}
```

61

## @IdClass

Quid des identifiants quand ceux-ci sont des objets

```
@Entity  
@IdClass(ParticipationId.class)  
public class Participation implements Serializable{  
    private static final long serialVersionUID = 6082551497084979589L;  
  
    @Id  
    @JoinColumn(name = "participation_training_id")  
    private Training training;  
  
    @Id  
    @JoinColumn(name = "participation_person_id")  
    private Person participant;  
  
    @Temporal(TemporalType.DATE)  
    private Date registration;  
  
    public Participation() {}  
  
    public Participation(Training training, Person participant) {  
        this.training = training;  
        this.participant = participant;  
    }  
}
```

62

## @IdClass

```
public class ParticipationId implements Serializable {  
    private static final long serialVersionUID = 8358358735060081078L;  
  
    private Long training;  
    private Long participant;  
  
    @Override  
    public boolean equals(Object o) {  
        if (this == o) return true;  
        if (!(o instanceof ParticipationId)) return false;  
        ParticipationId that = (ParticipationId) o;  
        return Objects.equals(getTraining(), that.getTraining()) &&  
            Objects.equals(getParticipant(), that.getParticipant());  
    }  
  
    @Override  
    public int hashCode() {  
        return Objects.hash(getTraining(), getParticipant());  
    }
```

Les types exposés dans la classe d'identification doivent être les même que les identifiants de chacune des classes

63

## Clé primaire

- 2 types de clé primaires
  - Clé naturelle (clé INSEE), constituée de donnée(s) métier
  - Clé substituée (Surrogate ou Substitute)
- Les clés substituées sont préférables aux clés composées
- 3 manières de les générer
  - Identity de colonne
  - Séquence SGBD
  - Table de séquence

64

## Identité de colonne

- Supporté par certaine base comme MS SQL Server & PostgreSQL
- Quand on l'utilise, la valeur générée peut ne pas être disponible avant sauvegarde en base

```
@Id  
@GeneratedValue(strategy = GenerationType.IDENTITY)  
@Column(name = "person_id")  
private Long id;
```

65

## @SequenceGenerator

```
@Target({TYPE, ElementType.METHOD, ElementType.FIELD})  
@Retention(RetentionPolicy.RUNTIME)  
public @interface SequenceGenerator {  
    java.lang.String name();  
    java.lang.String sequenceName() default "";  
    java.lang.String catalog() default "";  
    java.lang.String schema() default "";  
    int initialValue() default 1;  
    int allocationSize() default 50;  
}
```

67

## Séquence SGBD

- Nécessite la création d'une séquence en base de données et d'un SequenceGenerator (pas nécessairement dans le même entity)

```
@Id  
@SequenceGenerator(name="USER_SEQUENCE_GENERATOR",  
    sequenceName="USER_SEQUENCE", initialValue=1,  
    allocationSize=10)  
@GeneratedValue(strategy=GenerationType.SEQUENCE,  
    generator="USER_SEQUENCE_GENERATOR")  
@Column(name="USER_ID")  
private long id;
```

66

## Table de séquence

- Nécessite la création d'une table respectant un modèle précis

```
@Id  
@TableGenerator(name="USER_TABLE_GENERATOR",  
    table="SEQUENCE_GENERATOR_TABLE",  
    pkColumnName= "SEQUENCE_NAME",  
    valueColumnName= "SEQUENCE_VALUE",  
    pkColumnValue= "USER_SEQUENCE")  
@GeneratedValue(strategy=GenerationType.TABLE,  
    generator= "USER_TABLE_GENERATOR")  
@Column(name= "USER_ID")  
private long id;
```

68

## Table de séquence

- La table SEQUENCE\_GENERATOR\_TABLE

SEQUENCE_NAME	SEQUENCE_VALUE
USER_SEQUENCE	1
«autre séquence»	23

69

## Les relations

- Les entités peuvent être reliées à d'autres
- La cardinalité est gérée par annotations
  - @OneToOne (unidirectionnelle, défaut)
  - @OneToMany
  - @ManyToOne
  - @ManyToMany
- Elles peuvent être unidirectionnelles ou bidirectionnelles
- Les relations bidirectionnelles ont «Owning side» & «Inverse side»

71

## @TableGenerator

```
@Target({ElementType.TYPE, ElementType.METHOD, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface TableGenerator {
    java.lang.String name();
    java.lang.String table() default "";
    java.lang.String catalog() default "";
    java.lang.String schema() default "";
    java.lang.String pkColumnName() default "";
    java.lang.String valueColumnName() default "";
    java.lang.String pkColumnValue() default "";
    int initialValue() default 0;
    int allocationSize() default 50;
    javax.persistence.UniqueConstraint[] uniqueConstraints() default {};
    javax.persistence.Index[] indexes() default {};
}
```

70

## @OneToOne

```
@Entity
@Table(name = "PERSON")
public class Person {
    @Id
    @GeneratedValue(generator = "USER_SEQ")
    private Long id;
    @Column(nullable = false)
    private String name;
    private Address address;

    @Entity
    public class Address {
        @Id
        @GeneratedValue(generator = "ADDRESS_SEQ")
        private Long id;
        private String Street1;
        private String Street2;
        private String zipCode;
        private String city;
        private String country;
```

72

# @OneToOne

```
@Entity  
@Table(name = "PERSON")  
public class Person {  
    @Id  
    @GeneratedValue(generator = "USER_SEQ")  
    private Long id;  
    @Column(nullable = false)  
    private String name;  
    @OneToOne  
    private Address address;  
  
    @Entity  
    public class Address {  
        @Id  
        @GeneratedValue(generator = "ADDRESS_SEQ")  
        private Long id;  
        private String Street1;  
        private String Street2;  
        private String zipCode;  
        private String city;  
        private String country;
```

73

# @OneToOne

```
@Entity  
@Table(name = "PERSON")  
public class Person {  
    @Id  
    @GeneratedValue(generator = "USER_SEQ")  
    private Long id;  
    @Column(nullable = false)  
    private String name;  
    @OneToOne  
    @JoinColumn(name = "ADDRESS_FK", nullable = false)  
    private Address address;  
  
    public Person() {  
    }
```

74

# @OneToOne

```
@Target({ElementType.METHOD, ElementType.FIELD})  
@Retention(RetentionPolicy.RUNTIME)  
public @interface OneToOne {  
    java.lang.Class targetEntity() default void.class;  
    javax.persistence.CascadeType[] cascade() default {};  
    javax.persistence.FetchType fetch() default FetchType.EAGER;  
    boolean optional() default true;  
    java.lang.String mappedBy() default "";  
    boolean orphanRemoval() default false;  
}
```

75

# @OneToOne

- target Entity : type de classe pointée par la relation. Par défaut le “persistence provider“ utilise celui de l’attribut ou la valeur de retour du getter
- cascade : ce qu’il advient de la donnée pointée en cas de modification ou suppression de la relation
- fetch : façon dont les données sont peuplées
- optionnal : si la donnée référencée peut être nulle
- mappedBy : pour déclarer une relation bidirectionnelle, permet de positionner le côté “propriétaire“ de la relation

76

# @OneToOne

## Bidirectionnel

```
@Entity
@Table(name = "PERSON")
public class Person {
    @Id
    @GeneratedValue(generator = "USER_SEQ")
    private Long id;
    @Column(nullable = false)
    private String name;
    @OneToOne
    private Address address;

@Entity
public class Address {
    @Id
    @GeneratedValue(generator = "ADDRESS_SEQ")
    private Long id;
    private String Street1;
    .....
    private String country;
    @OneToOne(mappedBy = "address", optional = false)
    private Person person;
```

77

# @OneToOne

## Cas 1

```
@Entity
@Table(name = "PERSON")
public class Person {
    @Id
    @GeneratedValue(generator = "USER_SEQ")
    private Long id;
    @Column(nullable = false)
    private String name;
    @OneToOne
    @JoinColumn(name = "ADDRESS_FK", nullable = false)
    private Address address;

    public Person() {
    }
```

79

# mapping OneToOne

- Deux cas se présentent selon la table où se trouve la clé étrangère.
- Si A possède une référence sur B :
  - Cas 1 : la clé primaire de B est clé étrangère dans la table A
  - Cas 2 : la clé primaire de A est clé étrangère dans la table B

78

# @JoinColumn

```
@Target({ElementType.METHOD, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface JoinColumn {
    java.lang.String name() default "";
    java.lang.String referencedColumnName() default "";
    boolean unique() default false;
    boolean nullable() default true;
    boolean insertable() default true;
    boolean updatable() default true;
    java.lang.String columnDefinition() default "";
    java.lang.String table() default "";
}
```

80

# @OneToOne

Cas 2

```
@Entity
public class Person implements Serializable {
    private static final long serialVersionUID = 3457989253623908694L;

    /**
     * Identifiant
     */
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    /**
     * login
     */
    private String login;

    /**
     * Nom de famille
     */
    private String surname;

    /**
     * Prénom
     */
    private String firstname;

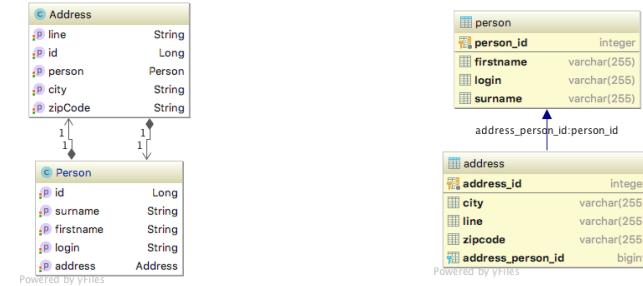
    @OneToOne
    @PrimaryKeyJoinColumn(name = "person_id", referencedColumnName = "address_person_id")
    private Address address;

    @ManyToOne(mappedBy = "participant", fetch = FetchType.LAZY)
    private Set<Person> trainingParticipationSet;
    public Person() {
    }
}
```

81

# @OneToOne

Cas 2



82

# @PrimaryKeyJoinColumn

```
@Target({ElementType.TYPE, ElementType.METHOD, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface PrimaryKeyJoinColumn {
    java.lang.String name() default "";
    java.lang.String referencedColumnName() default "";
    java.lang.String columnDefinition() default "";
}
```

83

# Les collections

```
@Entity
@Table(name = "PERSON")
public class Person {
    @Id
    @GeneratedValue(generator = "PERSON_SEQ")
    private Long id;
    .....
    private Set<Adhesion> adhesionSet;

    public Person() {
    }

    @Entity
    @Table(name = "ADHESION")
    public class Adhesion {
        @Id
        @GeneratedValue(generator = "ADHESION_SEQ")
        private Long id;
        @Temporal(TemporalType.TIMESTAMP)
        private Date date;
    }
}
```

84

# @OneToMany

```
@Entity  
@Table(name = "PERSON")  
public class Person {  
    @Id  
    @GeneratedValue(generator = "PERSON_SEQ")  
    private Long id;  
    .....  
    @OneToMany  
    @JoinTable(name = "ADHESIONJOINPERSON", joinColumns =  
    @JoinColumn(name = "PERSON_FK"), inverseJoinColumns =  
    @JoinColumn(name = "ADHESION_FK"))  
    private Set<Adhesion> adhesionSet;  
  
    public Person() {}  
}
```

85

# @OneToMany

Sans table de jointure

```
@Entity  
@Table(name = "PERSON")  
public class Person {  
    @Id  
    @GeneratedValue(generator = "PERSON_SEQ")  
    private Long id;  
    .....  
    @OneToMany(mappedBy = "person")  
    private Set<Adhesion> adhesionSet;  
    public Person() {}  
  
    @Entity  
    @Table(name = "ADHESION")  
    public class Adhesion {  
        @Id  
        @GeneratedValue(generator = "ADHESION_SEQ")  
        private Long id;  
        @Temporal(TemporalType.TIMESTAMP)  
        private Date date;  
        @ManyToOne  
        private Person person;  
  
        public Adhesion() {}  
    }
```

86

# @OneToMany & @ManyToOne

- Le One-to-many unidirectionnel n'est pas supporté dans la spécification. Bien que des "persistence provider" le supportent, il faut utiliser une table de relation avec `@JoinTable`
- Pas de `mappedBy` dans le `@ManyToOne` car c'est toujours cette table qui contient la clé étrangère
- Le `JoinColumn` peut pointer deux colonnes de la même table

87

# @ManyToMany

BiDirectionnel

```
@Entity  
@Table(name = "CATEGORY")  
public class Category {  
    @Id  
    @GeneratedValue(generator = "CATEGORY_SEQ")  
    @Column(name = "CATEGORY_ID")  
    private Long id;  
    @Column(nullable = false)  
    private String name;  
    @ManyToMany  
    private List<Event> eventList;  
  
    @Entity  
    @Table(name = "EVENTS")  
    public class Event {  
        @Id  
        @GeneratedValue(generator = "EVENT_SEQ")  
        @Column(name = "EVENT_ID")  
        private Long id;  
        @Column(nullable = false)  
        private String name;  
        @ManyToMany  
        private List<Category> category;
```

88

# @ManyToMany

Unidirectionnel

```
@Entity
@Table(name = "CATEGORY")
public class Category {
    @Id
    @GeneratedValue(generator = "CATEGORY_SEQ")
    @Column(name = "CATEGORY_ID")
    private Long id;
    @ManyToMany
    @JoinTable(
        name = "CATEGORY_EVENT",
        joinColumns = @JoinColumn(
            name = "CE_CATEGORY_ID",
            referencedColumnName = "CATEGORY_ID"
        ),
        inverseJoinColumns = @JoinColumn(
            name = "CE_EVENT_ID",
            referencedColumnName = "EVENT_ID"
        )
    )
    private List<Event> eventList;
```

89

# @ManyToMany

Bidirectionnel avec table de jointure

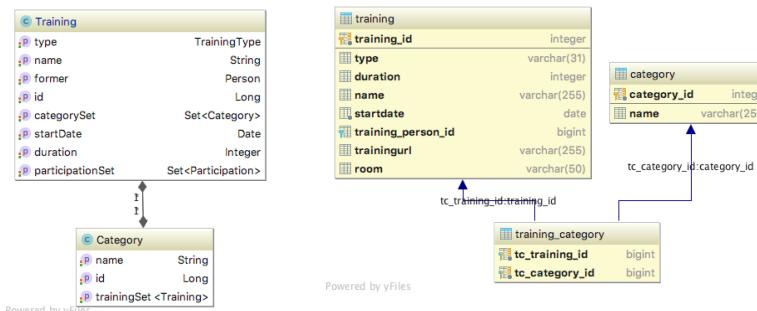
```
public class Training implements Serializable {
    private static final long serialVersionUID = -7688108012987555188L;
    @Id
    @GeneratedValue(strategy = IDENTITY)
    @Column(name = "training_id")
    private Long id;
    @ManyToMany(fetch = FetchType.LAZY)
    @JoinTable(name = "training_category"
        , joinColumns = {@JoinColumn(name = "tc_training_id", referencedColumnName = "training_id")}
        , inverseJoinColumns = {@JoinColumn(name = "tc_category_id", referencedColumnName = "category_id")})
    private Set<Category> categorySet;
}

@Entity
public class Category implements Serializable {
    private static final long serialVersionUID = 6132112112831771298L;
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "category_id")
    private Long id;
    private String name;
    @ManyToMany(fetch = FetchType.LAZY)
    @JoinTable(name = "training_category"
        , joinColumns = {@JoinColumn(name = "tc_category_id", referencedColumnName = "category_id")}
        , inverseJoinColumns = {@JoinColumn(name = "tc_training_id", referencedColumnName = "training_id")})
    private Set<Training> trainingSet;
}
```

90

# @ManyToMany

Bidirectionnel avec table de jointure



91

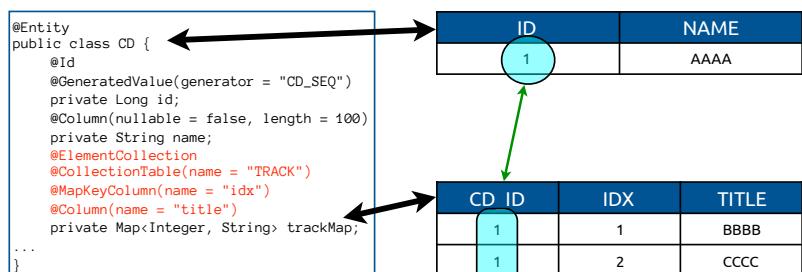
# Relation et FETCH

Relation	Stratégie FETCH
@OneToOne	EAGER
@ManyToOne	EAGER
@OneToMany	LAZY
@ManyToMany	LAZY

92

# Et les Map ?...

Petit rappel, une map = liaison clé/valeur



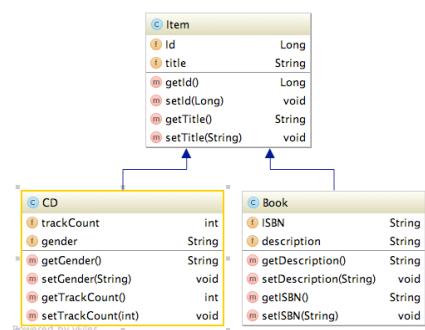
93

# L'héritage

- Le concept d'héritage n'existe pas dans le monde relationnel
- La transposition peut s'effectuer selon 3 stratégies
  - Une seule table
  - Tables jointes
  - Une table par classe

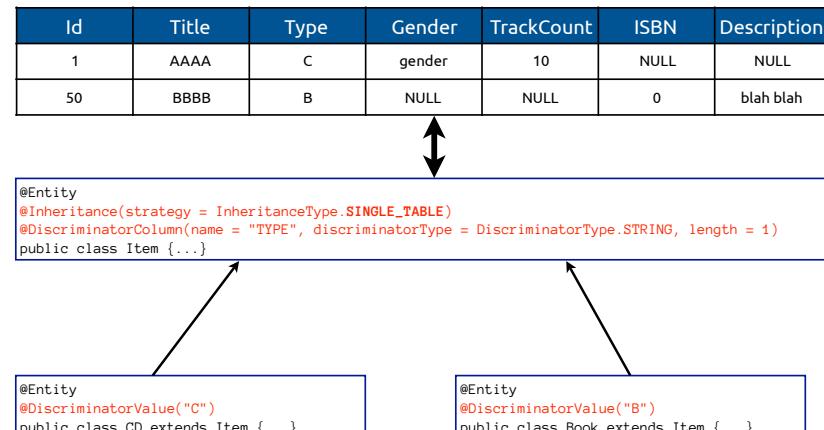
94

# Le modèle



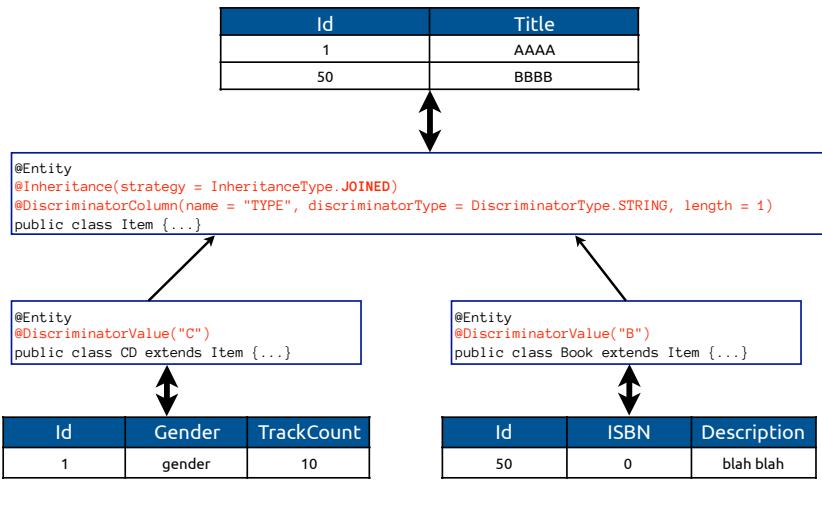
95

# Une table



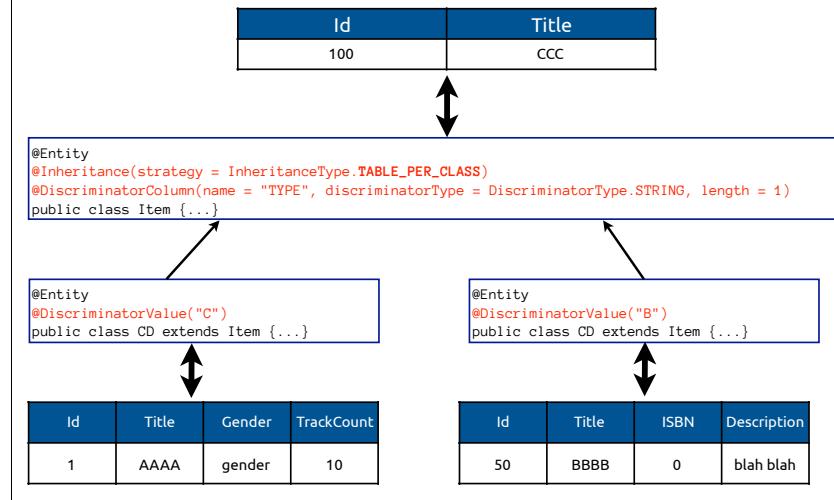
96

## Tables jointes



97

## Une table par classe

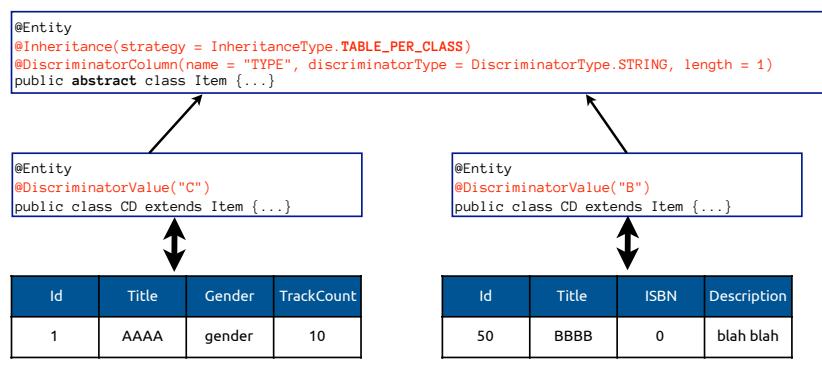


98

## Une table par classe



Classe parente abstraite, la table Item ne sera pas créée.



99

## L'héritage en bref

Fonction	Une table	Tables jointes	Une table par classe
Tables	•une seule table pour toute la hiérarchie •colonnes obligatoires peuvent être nullables •la table est modifiée en cas d'ajout de sous classes	•une pour la classe parent et une par sous-classe •les tables sont normalisées	une table par classes concrète dans la hiérarchie
utilise colonne discriminante	OUI	OUI	NON
SQL généré pour récupérer un entity	Select simple	Select avec Jointures	Select complexe (1 select par classe et union)
SQL généré pour insert et update	simple	Multiple en cascade	simple, par classe
Relation polymorphe	Bon	Bon	Mauvais
requête polymorphe	Bon	Bon	Mauvais
Implémentation JPA	Obligatoire	Obligatoire	Optionnel

100

# Persistance & JPA

101

## Pourquoi persister

- Les objets ne sont accessibles que lorsque la JVM fonctionne
- Si la JVM s'arrête, le GC nettoie la mémoire et les objets s'y trouvent
- Quelques objets doivent être persistés
- Stockés de manière permanente sur support magnétique, mémoire flash

103

## Persistance

- Les applications regroupent logique métier, IHM et Données
- Les données doivent être persistées
  - Fichier
  - base de données
- JPA permet de relier des objets à des bases relationnelles

102

## Comment persister en java

- Serialisation
- JDBC
- Object Relational Mapping (ORM)
  - JPA (EclipseLink)
  - Hibernate
  - Toplink

104

# Les avantages de JPA

## 2.1

- ORM : Mapping des objets
- Entity Manager...
- Langage de requête basé sur l'ORM (JPQL)
- Mécanisme de lock basé sur JTA
- Callback et listeners pour injecter une logique métier dans le cycle de vie.

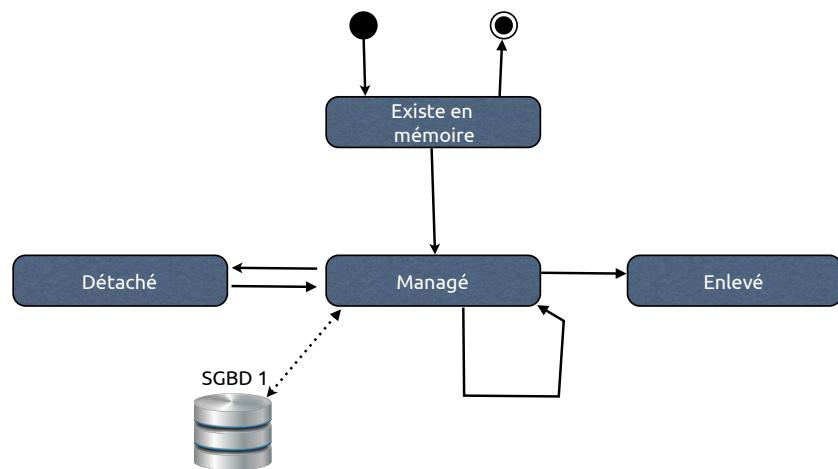
105

# EntityManager

- Il gère le cycle de vie des entités
- C'est la pièce centrale de JPA
- Il gère les requêtes sur les entités
- Il est garant des opérations CRUD

106

# Cycle de vie



107

# EntityManager

```
public interface EntityManager{  
    public void persist(Object entity);  
    public <T> T merge(T entity);  
    public void remove(Object entity);  
    public <T> T find(Class<T> entityClass, Object primaryKey);  
    public void flush();  
    public void setFlushMode(FlushModeType flushMode);  
    public FlushModeType getFlushMode();  
    public void refresh(Object entity);  
    public Query createQuery(String jpqlString);  
    public Query createNamedQuery(String name);  
    public Query createNativeQuery(String sqlString);  
    public Query createNativeQuery(String sqlString, Class resultClass);  
    public Query createNativeQuery(String sqlString, String resultSetMapping);  
    public void close();  
    public boolean isOpen();  
    public EntityTransaction getTransaction();  
    public void joinTransaction();  
    public void clear();  
}
```

108

# EntityManager

Il nécessite un fichier de configuration

META-INF/persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" 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/persistence_2_1.xsd">
    <persistence-unit name="blois-unitName" transaction-type="RESOURCE_LOCAL">
        <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
        <class>fr.blois.jee.jpa.td1.Activity</class>
        <properties>
            <property name="javax.persistence.jdbc.url" value="jdbc:derby://localhost:1527/jpa"/>
            <property name="javax.persistence.jdbc.driver"
value="org.apache.derby.jdbc.ClientDriver"/>
            <property name="javax.persistence.jdbc.user" value="blois"/>
            <property name="javax.persistence.jdbc.password" value="blois"/>
            <property name="eclipselink.ddl-generation" value="drop-and-create-tables"/>
        </properties>
    </persistence-unit>
</persistence>
```

109

## Obtenir le manager

- Géré par l'application

```
EntityManagerFactory entityManagerFactory =
Persistence.createEntityManagerFactory("blois-unitName");
EntityManager entityManager =
entityManagerFactory.createEntityManager();
...
entityManager.close();
entityManagerFactory.close();
```

- Géré par le container

```
@PersistenceContext(UnitName = "blois-unitName")
private EntityManager entityManager;

On peut aussi definir le scope transaction ou extended.
Extended, ne peut être activer qu'avec des StateFullSessionBean (SFSB)
```

111

# persistence.xml

- persistence-unit bloc de configuration
- provider, défini le vendeur (eclipseLink)
- class, défini les entités
- les propriétés
  - url d'accès à la base
  - driver de la base
  - Utilisateur / mot de passe
  - stratégie de génération (create, create and drop, none)

110

## @PersistenceContex

```
@Target({{TYPE, METHOD, FIELD}}) @Retention(RUNTIME)
public @interface PersistenceContext {
    String name() default "";
    String unitName() default "";
    PersistenceContextType type default TRANSACTION;
    PersistenceProperty[] properties() default {};
}
```

Attention, l'EntityManager n'est pas Thread safe, donc pas d'injection dans les classes comme les servlets

112

# Entité managée ?

- signifie que l'EntityManager s'assure que les données de l'entité sont synchronisées avec la base
- Quand l'entité devient managée, l'EntityManager synchronise son état avec la base de données
- Quand l'entité devient non-managée, l'EntityManager s'assure que les changements de données de l'entité sont répercutés en base

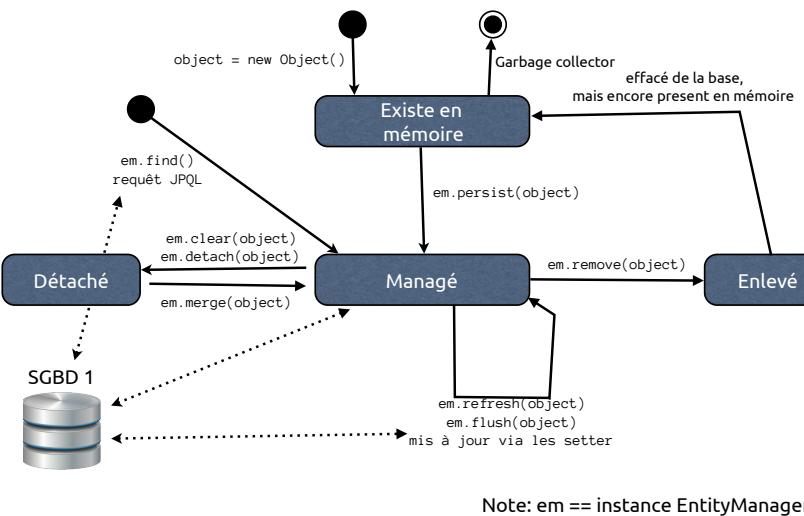
113

# Persister des entités

- `entityManager.persist(entity);`
- Si violation d'une contrainte d'intégrité : `PersistenceException` qui wrappe l'exception de la base de données
- Gestion automatique des clés selon la stratégie
- Problème pour la persistance de graphe d'objet : par défaut pas de persistance des objets en relation

115

# Cycle de vie



114

# Persister des entités

```
EntityTransaction entityTransaction = entityManager.getTransaction();
Person person = new Person("login", "surname", "firstname");
Address address = new Address("line", "zipcode", "city");
address.setPerson(person);
entityTransaction.begin();
entityManager.persist(person);
Long personId = person.getId();
entityManager.persist(address);
entityTransaction.commit();
person = entityManager.find(Person.class, personId);
```

Le manager cache toute action après le début de la transaction (`entityTransaction.begin()`);

116

# Charger une entité

- `entityManager.find(Entity.class, primaryKey);`
- Si les données ne sont pas trouvées en base, l'EntityManager renvoi null (ou un objet vide).
- Si Aucune transaction n'est démarrée, l'Objet renvoyé l'est en mode détaché
- Le chargement des objets (field) dépend de la stratégie de chargement (FETCH)
- Utilisation d'une requête JPQL

117

# Stratégies de chargement

- Détermine à quel moment les données seront récupérées
- 2 stratégies : LAZY & EAGER
  - EAGER: chargement immédiat (utilisation de requête join)
  - LAZY: chargement différé, exécuté à chaque demande de get (1 + N requêtes)

119

# Détachement

```
Person person = new Person("login", "surname", "firstname");
entityManager.persist(person);
Long personId = person.getId();
entityManager.detach(person);
Person samePerson = entityManager.find(Person.class, personId);
assertNotNull(samePerson);
assertEquals(person.getId(), samePerson.getId());
```

Utilisation de `entityManager.detach` pour que l'entité person ne soit plus Managé  
Le 2 objets person et samePerson sont égaux au niveau des données, mais il s'agit bien de 2 instances différentes

118

# Modifications

- Entité managé
  - Etat synchronisé via l'EntityManager (utilisation des setter, au sein d'une transaction si EM transactionnel)
- Entité non managée
  - Les données seront sauvegardées lors du rattachement
  - Si l'entité n'existe pas, une exception est levée (`IllegalArgumentException`)

120

# Suppression

```
Person person = new Person("toto");
Address address = new Address("rue", "ville", "00000", "pays");
person.setAddress(address);
entityTransaction.begin();
entityManager.persist(address);
entityManager.persist(person);
entityTransaction.commit();
entityTransaction.begin();
entityManager.remove(person);
entityTransaction.commit();
Assert.assertNotNull(person);
Person person1 = entityManager.find(Person.class, 1L);
Assert.assertNull(person1);
```

Les données “person” ont été supprimées de la base  
Cependant l’instance reste en mémoire jusqu’au passage du  
GarbageCollector

121

# Forcer les mises à jour

- l’EntityManager ne transfert pas les mises à jour immédiatement
- On peut cependant la forcer avec flush

122

## Forcer les mises à jour

```
Person person = new Person("toto");
Address address = new Address("rue", "ville", "00000", "pays");
person.setAddress(address);
entityTransaction.begin();
entityManager.persist(person);
entityManager.flush();
entityManager.persist(address);
entityTransaction.commit();
```

*Attention, cet exemple lève une IllegalStateException  
car la clé étrangère n'a pas été créée*

123

## Refresh data...

```
Person person = new Person("toto");
Address address = new Address("rue", "ville", "00000", "pays");
person.setAddress(address);
entityTransaction.begin();
entityManager.persist(person);
entityManager.persist(address);
entityTransaction.commit();
person.setName("titi");
entityManager.refresh(person);
Assert.assertEquals("toto", person.getName());
```

124

## Manipuler le PC

```
Person person = new Person("toto");
Address address = new Address("rue", "ville", "00000", "pays");
person.setAddress(address);
entityTransaction.begin();
entityManager.persist(person);
entityManager.persist(address);
entityTransaction.commit();
Assert.assertTrue(entityManager.contains(person));
entityManager.detach(person);
Assert.assertFalse(entityManager.contains(person));
```

125

## Evenements en cascade

Type	Description
PERSIST	Les relations sont persistées en même temps que la classe parent
REMOVE	Les relations sont supprimées en même temps que la classe parent
MERGE	La classe parent et les association sont soumises en même temps aux opérations de rattachement
REFRESH	Classes et associations sont rafraîchis au même moment
DETACH	Classes et associations sont détachées au même moment
ALL	Association de toutes les conditions précédentes

127

## Rattachement

```
Person person = new Person("toto");
Address address = new Address("rue", "ville", "00000", "pays");
person.setAddress(address);
entityTransaction.begin();
entityManager.persist(person);
entityManager.persist(address);
entityTransaction.commit();
entityManager.clear();
person.setName("titi");
entityTransaction.begin();
entityManager.merge(person);
entityTransaction.commit();
```

126

## Evenements en cascade

```
@Entity
@Table(name = "PERSON")
public class Person {
    @Id
    @GeneratedValue(generator = "PERSON_SEQ")
    @Column(name = "PERSON_ID")
    private Long id;
    @Column(nullable = false)
    private String name;
    @OneToOne
    @JoinColumn(name = "ADDRESS_FK", nullable = false)
    private Address address;
}

entityTransaction.begin();
entityManager.persist(person);
entityManager.persist(address);
entityTransaction.commit();
```

128

## Evenements en cascade

```
@Entity  
@Table(name = "PERSON")  
public class Person {  
    @Id  
    @GeneratedValue(generator = "PERSON_SEQ")  
    @Column(name = "PERSON_ID")  
    private Long id;  
    @Column(nullable = false)  
    private String name;  
    @OneToOne(  
        fetch = FetchType.LAZY,  
        cascade = {CascadeType.PERSIST, CascadeType.MERGE}  
    )  
    @JoinColumn(name = "ADDRESS_FK", nullable = false)  
    private Address address;  
  
    entityTransaction.begin();  
    entityManager.persist(person);  
    entityTransaction.commit();
```

129

## Requeter une entité

- EntityManager gère les opérations de type CRUD
- Les opérations CRUD ne suffisent pas forcément
- Besoin d'un langage de requête complexe
- avec une vue objet (pas de vue SGDB)
- Utilisation de JPQL
- *On peut aussi utiliser le SQL*

130

## JPQL

- Langage de requête JPA
- Converti ses requêtes en SQL via le JPQL Processor Query



Si les requêtes sont exécutées hors transaction, les entités sont détachées

131

## Utilisation JPQL

```
Person person = new Person("toto");  
Address address = new Address("rue", "ville", "00000", "pays");  
person.setAddress(address);  
entityTransaction.begin();  
entityManager.persist(person);  
entityTransaction.commit();  
entityManager.clear();  
Query query =  
    entityManager.createQuery("select p from Person p");  
List<Person> personList = query.getResultList();  
Assert.assertTrue(personList.size() == 1);  
Assert.assertEquals(person, personList.get(0));
```

132

## Requêtes nommées

- Centraliser les requêtes, garder la logique métier claire
- Maintenance plus aisée
- Amélioration des performances



Attention, une requête nommée est liée au scope du PersistenceUnit, **son nom doit être unique.**

133

## Requêtes paramétrées

- Possibilité de passer un paramètre par son index

```
Query query = entityManager.createQuery(  
    "select p from Person p where p.name = ?1");  
query.setParameter(1, "toto");
```

- Possibilité de passer un paramètre par un alias

```
Query query = entityManager.createQuery(  
    "select p from Person p where p.name = :nameParameter");  
query.setParameter("nameParameter", "toto");
```

135

## Utilisation JPQL

```
@Entity  
@Table(name = "PERSON")  
@NamedQueries(  
    @NamedQuery(name = "Person_findAll", query = "select p from Person  
p")  
)  
public class Person { ... }  
  
Person person = new Person("toto");  
Address address = new Address("rue", "ville", "00000", "pays");  
person.setAddress(address);  
entityTransaction.begin();  
entityManager.persist(person);  
entityTransaction.commit();  
entityManager.clear();  
Query query =  
    entityManager.createNamedQuery("Person_findAll");  
List<Person> personList = query.getResultList();  
Assert.assertTrue(personList.size() == 1);  
Assert.assertEquals(person, personList.get(0));
```

134

## JPQL pour charger

- Charger une liste d'entités

```
List personList = query.getResultList();  
• aucun résultat == liste vide  
• Possibilité de pagination via l'API Query
```

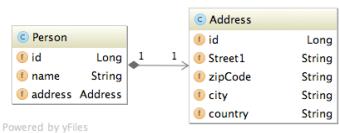
- Charger un résultat unique

```
Person personFound = (Person)query.getSingleResult();  
• Lève des exceptions  
NonUniqueResultException et  
NoResultException qui sont des exceptions  
RunTime
```

136

# JPQL et les relations

Sélection d'un élément via l'arbre de relation objet...



```
Person person = new Person("toto");
Address address = new Address("rue", "ville", "00000", "pays");
person.setAddress(address);
.....
Query query = entityManager.createQuery("select p from Person p
where p.address.country = :countryParameter");
query.setParameter("countryParameter", "pays");
Person personFound = (Person)query.getSingleResult();
```

137

## Opérateurs

Type	Opérateurs
Navigation	.
Signe Unaire	+ -
Arithmétique	/ * - +
Relationnel	<, >, =, <=, >=, [NOT] BETWEEN, [NOT] LIKE, [NOT] IN, IS [NOT] NULL, IS [NOT] EMPTY, [NOT] MEMBER [OF]
Logique	NOT, AND, OR

139

# Mots clé

- Statement et clause :** SELECT, UPDATE, DELETE, FROM, WHERE, GROUP, HAVING, ORDER, BY, ASC, DESC
- Jointures :** JOIN, OUTER, INNER, LEFT, FETCH
- Conditions et opérateurs :** DISTINCT, OBJECT, NULL, TRUE, FALSE, NOT, AND, OR, BETWEEN, LIKE, IN, AS, UNKNOWN, EMPTY, MEMBER, OF, IS, NEW, EXISTS, ALL, ANY, SOME
- Fonctions :** AVG, MAX, MIN, SUM, COUNT, MOD, UPPER, LOWER, TRIM, POSITION, CHARACTER\_LENGTH, CHAR\_LENGTH, BIT\_LENGTH, CURRENT\_TIME, CURRENT\_DATE, CURRENT\_TIMESTAMP

138

## Structure d'une requête

```
SELECT [DISTINCT] <expression de selection> [[AS] <alias>]
FROM <clause from (classe)>
[WHERE <conditions>]
[ORDER BY <clause de tri>]
[GROUP BY <clause de regroupement>]
[HAVING <clause de « possession »>]
```

140

## Jointure

- Les jointures suivent la même logique qu'en SQL
- INNER JOIN
- LEFT OUTER JOIN

141

## Opérations de masse

- Les actions en masse sont possibles
- Mise à jour globale (ou sur critères)
- Effacement globale (ou sur critères)

142

## Utilisation SQL

### Requête native

- Possibilité d'utiliser des requêtes SQL

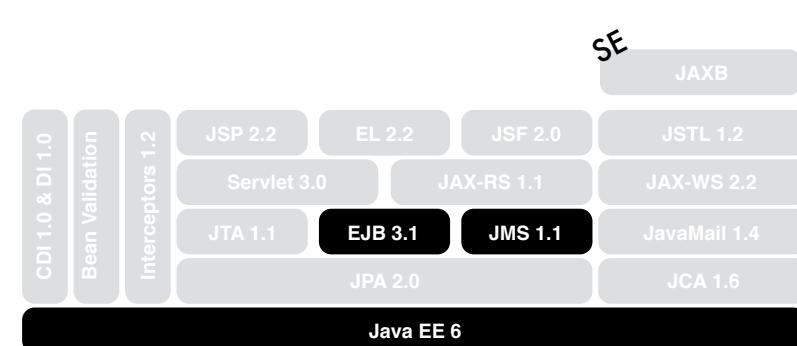
```
Query query = entityManager.createNativeQuery("SELECT * FROM PERSON");
```

- Utilisation de particularité de la base
- Portabilité moyenne
- Utilisation de ResultSetMapping pour le résultat
- Accès aux procédures stockées, par exemple

143

## JPA

### Les entités et JPA



144

## EJB == Logique métier

- La couche de persistance n'est pas appropriée pour la logique métier
- La couche de présentation ne doit pas exécuter la logique métier
- Besoin de sécurité et de transaction
- Interaction avec des services extérieurs

Besoin d'une couche métier !

145

## un EJB...

```
@Stateless  
public class UserEJB {  
  
    @PersistenceContext  
    EntityManager entityManager;  
  
    public User findUserById(Long id) {  
        return entityManager.find(User.class, id);  
    }  
}
```

147

## Les types d'EJB

- Stateless: Gestion sans état. **Session bean**
- Stateful: Gestion conversationnelle
- Singleton
- Message Driven bean: interaction avec JMS

146

## Anatomie d'un EJB

- C'est un POJO
- Annotation @Stateless, @Statefull, @Singleton, @MessageDriven (ou un descripteur XML)
- Il peut avoir plusieurs interfaces.
- Constructeur sans arguments
- Classe non finale, ni abstraite

148

## Comment appeler un EJB

```
public class Main {  
  
    @EJB  
    private static UserEJB userEJB;  
  
    public static void main(String... args) {  
        User user = userEJB.findUserById(1L);  
    }  
}
```

149

## Intégration

- Session Beans, MDBs, Entities
- Aide à mettre les composants en relation par la configuration plutôt que le code, par Injection de dépendance (Dependency Injection DI) et Lookup.

151

## Les services de la couche métier

- Intégration
- Sécurité
- intercepteurs
- Cycle de vie et pooling
- Accès distant
- Gestion de l'état
- Messaging
- Transaction
- Multi thread
- Web service
- Invocation asynchrone
- Injection de dépendances

150

## Pooling

- SLSBs, MDBs
- Création d'un lot d'instance des Beans gérés par le container. Le container n'autorise qu'un seul client à accéder à une instance. Après utilisation l'instance retourne dans le pool.

152

## Threads

- Session Beans, MDBs
- Le container assure l'accès à une instance par un seul client. Pas de gestion de la concurrence.
- Depuis JEE 7, dans certains cas le multitread est possible, mais interdit dans les versions antérieures

153

## Gestion de l'état

- SFSB (Statefull Session Bean)
- Gestion automatique de l'état par le container. Il gère la persistance des SFSB et l'association des instances avec les utilisateurs

154

## Messaging

- MDBs (Message Driven Beans)
- Simplification extrême de la création de composants <sup>8</sup>messaging-aware<sup>8</sup>: écoute un canal de communication et réagit à l<sup>9</sup>arrivée de messages

155

## Transaction

- Session Beans, MDBs
- Possibilité d'utiliser la configuration pour déléguer la gestion des transactions au container d'EJB.

156

## Sécurité

- Session Beans
- Possibilité d'utiliser la configuration pour interagir avec JAAS (Java Authentication and Authorization Service)

157

## intercepteur

- Session Beans, MDBs
- Externalise la gestion des problèmes transverses des applications (logging, auditing...)
- Version simplifiée de l'AOP (Aspect Oriented Programming)

158

## Accès distant

- Session Beans
- Accès distant (RMI) sans avoir de code à écrire
- Possibilité d'injection de dépendance permettant une utilisation comme si on était dans le container

159

## Web services

- SLSBs
- Minimalise les changements de code pour rendre un Stateless Session Bean accessible par Service Web

160

## Invocation d'un EJB

- JNDI Lookup permet la récupération d'objet dans l'arbre JNDI du serveur
  - Problème : couplage du code avec le serveur, "boilerplate code"
- Injection Dépendance aussi simple qu'une annotation

161

## Injection de dependance

```
@Stateless  
public class UserEJB {  
    @PersistenceContext  
    EntityManager entityManager;  
    public User findUserById(Long id) {  
        return entityManager.find(User.class, id);  
    }  
}  
  
public class Main {  
    @EJB  
    private static UserEJB userEJB;  
    public static void main(String... args) {  
        User user = userEJB.findUserById(1L);  
    }  
}
```

163

## Lookup JNDI

```
@Stateless  
@EJB(name = "UserEJBSERVICE")  
public class UserEJB {  
    @PersistenceContext  
    EntityManager entityManager;  
    public User findUserById(Long id) {  
        return entityManager.find(User.class, id);  
    }  
}  
  
public void MainString... args() {  
    try {  
        InitialContext initialContext = new InitialContext();  
        UserEJB userEJB = (UserEJB) initialContext.lookup("java:comp/env/UserEJBSERVICE");  
        User user = userEJB.findUserById(1L);  
    } catch(NamingException e) {  
        // gestion exception  
    }  
}
```

162

## @EJB

```
@Target({TYPE, METHOD, FIELD})  
@Retention(RUNTIME)  
public @interface EJB{  
    String name() default IJ;  
    Class beanInterface() default Object.class;  
    String beanName() default IJ;  
}
```

164

## @EJB

- name : Nom utilisé pour lier à l'arbre JNDI.
- beanInterface : Interface business utilisée pour accéder à l'EJB
- beanName : Distinction si plusieurs EJBs implémentent la même interface business.

165

## Les Bean de session

167

## Annotation ou XML

- Question de goût.
- Annotations : concis, proche du code
  - Beaucoup ont des valeurs par défaut
- XML : verbeux mais modifiable sans recompilation
- XML permet un surcharge des annotations

166

## Quelques définitions

- Session : connexion entre un client et un serveur qui dure une période de temps finie
- Client : ligne de commande, composant web (servlet, JSP, JSF,...), application Desktop, voir application .NET via des web services

168

## Les outils des bean de session

- Les EJB fournissent des services aux développeurs
  - Concurrence et Thread Safety
  - Remoting et web services
  - Transaction et sécurité
  - Timers et intercepteurs

169

## EJB et interfaces

```
@Stateless  
public class HelloToTheWorld {  
    public String sayBonjour(String name) {  
        return "bonjour " + name;  
    }  
}
```

171

## Les <sup>8</sup>session beans<sup>8</sup>

- Stateless: littéralement sans état conversationnel, Utilisé pour réaliser des tâches avec un simple appel de méthode.
- Statefull: Maintient un état (et les données associées) avec un client spécifique. Utilisé pour réaliser des actions en plusieurs étapes.
- Singleton: design pattern du singleton. C'est le container qui s'assure de l'unicité de cet EJB.

170

## EJB et interfaces

```
@Stateless @LocalBean  
public class HelloToTheWorld implements HelloToTheWorldLocal {  
    public String sayBonjour(String name) {  
        return "bonjour " + name;  
    }  
    @Override  
    public String sayGutenTag(String name) {  
        return "guten tag " + name;  
    }  
}
```

```
@Local  
public interface HelloToTheWorldLocal {  
    String sayGutenTag(String name);  
}
```

172

## EJB et interfaces

```
@Stateless @LocalBean
public class HelloToTheWorld implements HelloToTheWorldLocal, HelloToTheWorldRemote {
    public String sayBonjour(String name) {
        return "bonjour " + name;
    }
    @Override
    public String sayGutenTag(String name) {
        return "guten tag " + name;
    }
    @Override
    public String sayHello(String name) {
        return "hello " + name;
    }
}
```

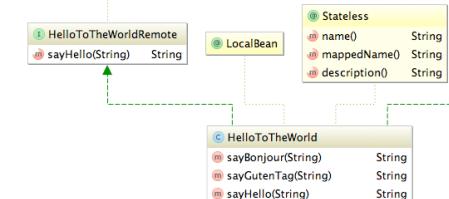
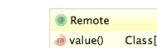
```
@Local
public interface HelloToTheWorldLocal {
    String sayGutenTag(String name);
}
```

```
@Remote
public interface HelloToTheWorldRemote {
    String sayHello(String name);
}
```

173

## EJB et interfaces

```
@Stateless @LocalBean
public class HelloToTheWorld implements HelloToTheWorldLocal, HelloToTheWorldRemote {
    public String sayBonjour(String name) {
        return "bonjour " + name;
    }
    @Override
    public String sayGutenTag(String name) {
        return "guten tag " + name;
    }
    @Override
    public String sayHello(String name) {
    }
}
```



174

## EJB et interfaces (alt)

```
@Stateless @LocalBean
@Local(value = HelloToTheWorldLocal.class)
@Remote(value = HelloToTheWorldRemote.class)
public class HelloToTheWorld implements HelloToTheWorldLocal, HelloToTheWorldRemote {
    public String sayBonjour(String name) {
        return "bonjour " + name;
    }
    @Override
    public String sayGutenTag(String name) {
        return "guten tag " + name;
    }
    @Override
    public String sayHello(String name) {
        return "hello " + name;
    }
}
```

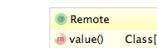
```
public interface HelloToTheWorldLocal {
    String sayGutenTag(String name);
}
```

```
public interface HelloToTheWorldRemote {
    String sayHello(String name);
}
```

175

## EJB, interfaces & appel

```
public class CallHelloToTheWorld {
    @EJB
    private HelloToTheWorld helloToTheWorld;
    @EJB
    private HelloToTheWorldLocal helloToTheWorldLocal;
    @EJB
    private HelloToTheWorldRemote helloToTheWorldRemote;
}
```



176

## EJB et interfaces (bonus)

```
@Stateless @LocalBean
public class HelloToTheWorld
    implements HelloToTheWorldRemote, HelloToTheWorldLocal, HelloToTheWorldSOAP, HelloToTheWorldRest {
    public String sayBonjour(String name) {
        return "bonjour " + name;
    }
    @Override
    public String sayGutenTag(String name) {
        return "guten tag " + name;
    }
    @Override
    public String sayHello(String name) {
        return "hello " + name;
    }
    @Override
    public String sayHelloSOAP() {
        return "hello through SOAP";
    }
    @Override
    public String sayHelloREST() {
        return "hello through REST";
    }
}
```

```
@Remote
public interface HelloToTheWorldRemote {
    String sayHello(String name);
}
```

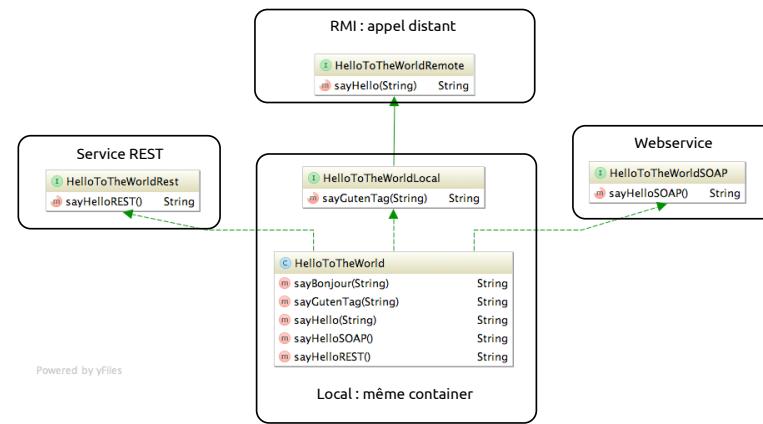
```
@Local
public interface HelloToTheWorldLocal
    extends HelloToTheWorldRemote {
    String sayGutenTag(String name);
}
```

```
@WebService
public interface HelloToTheWorldSOAP {
    String sayHelloSOAP();
}
```

```
@Path("/hello")
public interface HelloToTheWorldRest {
    String sayHelloREST();
}
```

177

## EJB et interfaces (bonus)



178

## Règles de programmation

- Au moins une interface métier (business)
- Classe concrète
- Constructeur sans argument
- Plusieurs annotations sur une interface, impossible, utilisation de l'héritage
- Les règles d'héritage OO s'appliquent

179

## Règles de programmation

- Héritage des annotations pour DI et lifecycle callback
- Les méthodes business ne doivent pas démarrer par “ejb”
- Les méthodes business doivent être public, non final et non static
- Si les méthodes sont dans l’interface remote, les arguments et le type de retour doivent implémenter `java.io.Serializable`

180

## Stateless ou Stateful ?

- Cinématique ayant besoin d'un état conversationnel (gestion de caddie, formulaire sur plusieurs pages...)
- Stateful bean permet de gérer cette conservation des données en session sur le serveur, plus simplement que la session HTTP

181

## Stateless

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface Stateless {
    java.lang.String name() default "";
    java.lang.String mappedName() default "";
    java.lang.String description() default "";
}
```

182

## Stateless

```
@Stateless @LocalBean
public class HelloToTheWorld implements HelloToTheWorldLocal {
    public String sayBonjour(String name) {
        return "bonjour " + name;
    }
    @Override
    public String sayGutenTag(String name) {
        return "guten tag " + name;
    }
}
```

183

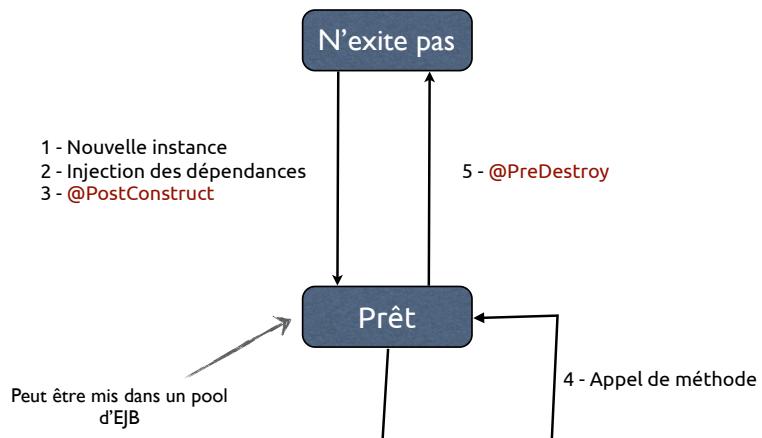
## Stateless

- name : Nom du Bean. Certains containers l'utilisent pour lier l'EJB dans l'arbre JNDI. Par défaut le nom de la classe
- mappedName : vendeur spécifique. Utilisé par certains containers pour lier l'EJB dans l'arbre JNDI
- description : pour la console d'administration

184

## Stateless

### Cycle de vie



185

## Singleton

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface Singleton {
    java.lang.String name() default "";
    java.lang.String mappedName() default "";
    java.lang.String description() default "";
}
```

186

## Singleton

```
@Singleton
public class AddressCache {
    public Address getAddressFromCache(long id) {return address;}
    public void putAddressToCache(Address address) {}
    public void removeAddressFromCache(long id) {}
}
```

187

## Singleton

```
@Singleton
@Startup
public class AddressCache {

    @PostConstruct
    private void addressCacheInitialization() {
        // Initialisation longue
    }
    public Address getAddressFromCache(long id) {return address;}
    public void putAddressFromCache(Address address) {}
    public void removeAddressFromCache(long id) {}
}
```

188

# Singleton

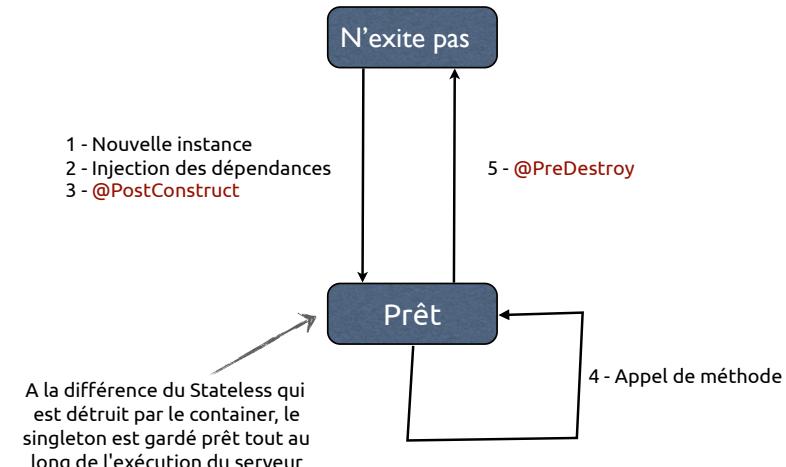
## Enchainement

```
@Singleton  
public class ZipcodeCache {  
}  
@Singleton  
@Startup  
@DependsOn("CityCache", "ZipcodeCache")  
public class AddressCache {  
    @PostConstruct  
    private void addressCacheInitialization() {  
        // Initialisation longue  
    }  
    public Address getAddressFromCache(long id) {return address;}  
    public void putAddressFromCache(Address address) {}  
    public void removeAddressFromCache(long id) {}  
}
```

189

# Singleton

## Cycle de vie



190

# Singleton

## Concurrence

- Une instance, plusieurs clients
  - Problèmes d'accès concurrent
  - **@ConcurrencyManagement**
  - Concurrence gérée par le container
    - Container-Management Concurrency CMC
  - Concurrence gérée par le bean
    - Bean-Managed Concurrency BMC

191

# Singleton

## Concurrence gérée par le container

- Le verrouillage peut être dirigé **@Lock**
  - **@Lock(LockType.READ)** - verrou partagé
  - **@Lock(LockType.WRITE)** - verrou exclusif
- Déclaration au niveau de la classe et des méthodes

192

# Singleton

Concurrence gérée par le container

```
@Singleton  
@Lock(LockType.READ)  
public class AddressCache {  
    public Address getAddressFromCache(long id) {  
        return address;  
    }  
    @Lock(LockType.WRITE)  
    @AccessTimeout(2000)  
    public void putAddressToCache(Address address) {  
    }  
    public void removeAddressFromCache(long id) {  
    }  
}
```

193

# Singleton

Concurrence gérée par le bean

- Le développeur doit gérer les accès concurrentiels
- Utilisation de synchronized & volatile

194

# Singleton

Concurrence gérée par le bean

```
@Singleton  
@ConcurrencyManagement(ConcurrencyManagementType.BEAN)  
public class AddressCache {  
    public synchronized Address getAddressFromCache(long id) {  
        return null;  
    }  
    public synchronized void putAddressToCache(Address address) {  
    }  
    public void removeAddressFromCache(long id) {  
    }  
}
```

195

# Stateful

```
@Stateful  
@StatefulTimeout(value = 30, unit = TimeUnit.SECONDS)  
public class HelloToWorld implements HelloToWorldLocal {  
    public String sayBonjour(String name) {  
        return "bonjour " + name;  
    }  
    @Override  
    public String sayGutenTag(String name) {  
        return "guten tag " + name;  
    }  
}
```

196

# Stateful

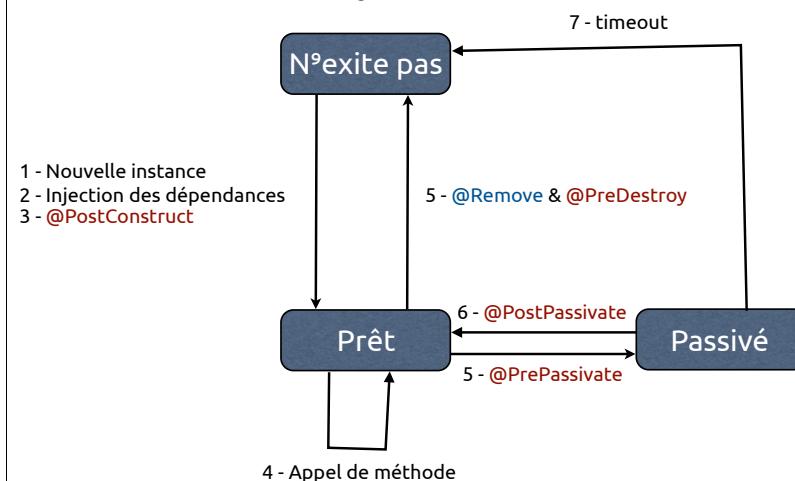
## Règles de programmation

- Les attributs doivent être des primitives ou implémenter `java.io.Serializable` pour permettre la passivation
- Prévoir une méthode de destruction pour libérer les ressources
- Prendre en compte du cycle de vie différent (callback)

197

# Stateful

## Cycle de vie



198

# Stateful

## Passivation

- Sérialisation de l'EJB sur le disque lorsqu'il est trop longtemps sans servir
- Le sens contraire : Activation
- deux callbacks : `@PrePassivate` `@PostActivate`, il peut s'agir des mêmes méthodes que pour `@PostConstruct` et `@PreDestroy`

199

# Stateful

## `@Remove`

- Annotation sur une ou plusieurs méthodes de l'EJB
- Indique que l'EJB doit être détruit après exécution de la méthode
- Permet libération des ressources de l'EJB sans attente du timeout et évite passivation/activation excessives

200

# Stateful

Règle pour l'appel d'un EJB Stateful

- Pas d'injection de dépendance d'un SFSB dans un SLSB
- Injection d'un SFSB dans une servlet, même instance pour tous les clients

201

# Stateful

Performance

- Coût d'un EJB Statefull
  - Occupation mémoire / disque
  - Problème de réPLICATION en réseau (cluster) et donc coût réseau
- Attention aux objets référencés (si possible utiliser des identifiants)
- Attention à la configuration (nom de SFSB max actif et timeout)
- Ne pas oublier @Remove

202

## Différence SFSB / SLSB

Fonctionnalité	Stateless	Stateful
État conversationnel	NON	OUI
Problème de performance	Selon les méthodes	Possible
lifecycle callback	@PostConstruct & @PreDestroy	@PostConstruct, @PreDestroy, @PrePassivated, @PostPassivated
Timer	OUI	NON
Synchronisation de la session	NON	OUI
WebService	OUI	NON
Pool	OUI	NON
Extended Persistence Context	NON	OUI

203

## EJB & Transaction

204

# Transaction

- Les données sont fondamentales
- Elles doivent être précises quelles que soit les opérations effectuées
- Y compris lors d'un accès concurrentiel
- Les transactions assurent un état consistant
- Une série d'opérations a besoin d'être exécutée comme une simple opération

205

# ACID

- **Atomicity** : tout ou rien, la transaction se termine soit par un commit soit un rollback
- **Consistency** : le système est dans un état consistant avant la transaction et dans un état consistant après, quel que soit le devenir de la transaction.
- **Isolation** : Les changements d'une transaction ne sont pas visibles à une autre transaction avant le commit
- **Durability** : les données commitées sont permanentes et survivent à un crash système

206

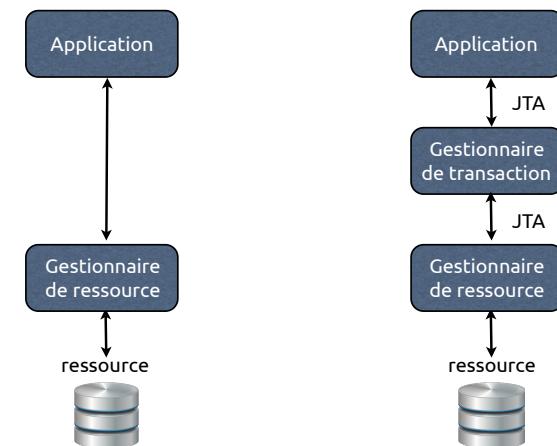
# support transaction

- Les EJB sont transactionnelles par défaut
- Les outils du framework vous aident :
  - Transaction manager
  - Resource manager
- Utilisation JTA
- 2 gestions des transactions
  - Container-Managed-Transaction (CMT)
  - Bean-Managed-Transaction (BMT)

207

# Transaction locale

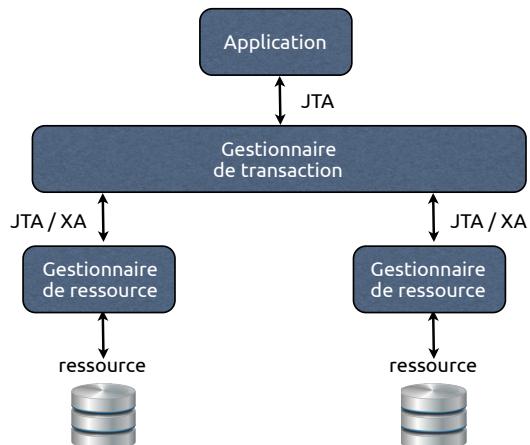
une seule ressource transactionnelle



208

# Transactions Distribuées

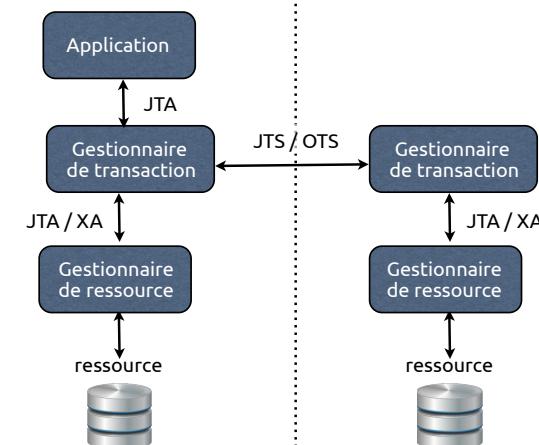
XA



209

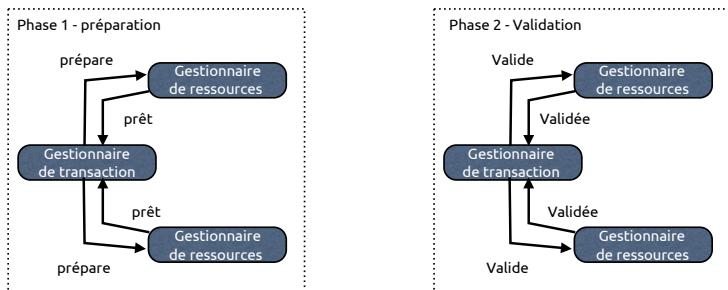
# Transactions Distribuées

XA via le reseau



210

# Validation en 2 phases



211

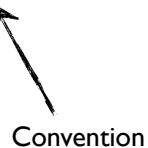
# Container-Managed-Transaction

```
@Stateless  
@TransactionManagement(TransactionManagementType.CONTAINER)  
public class TaskToDoEJB {  
    @PersistenceContext  
    private EntityManager entityManager;  
  
    @EJB  
    private ToDoListEJB toDoListEJB;  
  
    public Task createTask(Task task) {  
        entityManager.persist(task);  
        toDoListEJB.addTask(task);  
        return task;  
    }  
}
```

212

## Container-Managed-Transaction

```
@Stateless  
@TransactionManagement(TransactionManagementType.CONTAINER)  
public class TaskToDoEJB {  
    @PersistenceContext  
    private EntityManager entityManager;  
  
    @EJB  
    private ToDoListEJB toDoListEJB;  
  
    public Task createTask(Task task) {  
        entityManager.persist(task);  
        toDoListEJB.addTask(task);  
        return task;  
    }  
}
```



Convention

213

## Attributs de transaction

Gestion des transactions par le container  
@TransactionAttribute

@TransactionAttribute	Dans une transaction	hors Transaction
REQUIRED	Rattrape la transaction du client (appelant)	Création d'une nouvelle transaction
REQUIRES_NEW	Création d'une nouvelle transaction et suspend la transaction du client	Création d'une nouvelle transaction
SUPPORTS	Rattrape la transaction du client (appelant)	Pas de transaction
MANDATORY	Rattrape la transaction du client (appelant)	déclenche une EJBTransactionRequiredException
NOT_SUPPORTED	La transaction du client est suspendue et aucune transaction est créée	Pas de transaction
NEVER	déclenche une EJBException	Pas de transaction

214

## Annulation de transaction

- On peut noter un transaction en rollback en utilisant la méthode SessionContext.setRollbackOnly()
- l'appel de cette méthode hors contexte transactionnel ou dans un contexte BMT lève une IllegalStateException

```
@PersistenceContext  
private EntityManager entityManager;  
@Resource  
private SessionContext sessionContext;  
@TransactionAttribute(TransactionAttributeType.MANDATORY)  
public Task createTask(Task task) {  
    entityManager.persist(task);  
    toDoListEJB.addTask(task);  
    sessionContext.setRollbackOnly();  
    return task;  
}
```

215

## Transactions et exceptions

- par défaut toutes les “checked Exception“ sont gérées par le client et se voit ajouter le @ApplicationException
- les “runtime Exception“ sont wrapées dans des EJBException
- Par défaut, toutes les exceptions entraînent un rollback
- par contre, la gestion du rollback peut être modérée

```
@ApplicationException(rollback = false)  
public class TaskAlreadyDoneException  
extends Exception {  
}
```

216

## Bean-Managed-Transaction

```
@Resource  
private UserTransaction userTransaction;  
public Task createTask(Task task) {  
    try {  
        userTransaction.begin();  
        entityManager.persist(task);  
        ToDoListEJB.addTask(task);  
        if (noError) {  
            userTransaction.commit();  
        } else {  
            userTransaction.rollback();  
        }  
    } catch (Exception e) {  
        try {  
            userTransaction.setRollbackOnly();  
        } catch (SystemException e1) {  
            // Gestion erreur  
        }  
        // gestion erreur  
    }  
    return task;  
}
```

217

## @UserTransaction

```
package javax.transaction;  
  
public interface UserTransaction {  
    void begin() throws NotSupportedException, SystemException;  
  
    void commit() throws RollbackException, HeuristicMixedException, HeuristicRollbackException,  
SecurityException, IllegalStateException, SystemException;  
  
    void rollback() throws IllegalStateException, SecurityException, SystemException;  
  
    void setRollbackOnly() throws IllegalStateException, SystemException;  
  
    int getStatus() throws SystemException;  
  
    void setTransactionTimeout(int i) throws SystemException;  
}
```

218

## @UserTransaction et status

- Interface regroupant les valeurs retournées par UserTransaction.getStatus()
- Les valeurs possibles :
  - STATUS\_ACTIVE
  - STATUS\_MARKED\_ROLLBACK
  - STATUS\_PREPARED
  - STATUS\_COMMITTED
  - STATUS\_ROLLEDBACK
  - STATUS\_UNKNOWN
  - STATUS\_NO\_TRANSACTION
  - STATUS\_PREPARING
  - STATUS\_COMMITTING
  - STATUS\_ROLLING\_BACK

219

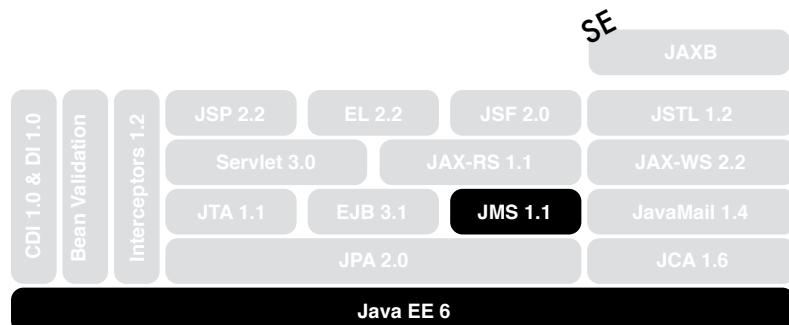
## Avantages & inconvénients BMT

- Avantages
  - gestion plus fine des limites de la transaction
  - permet maintien de transactions entre les appels pour un SFSB
- Inconvénients
  - code plus verbeux et susceptible d'erreur
  - ne peut joindre une transaction courante

220

# JMS

## Message Driven Bean



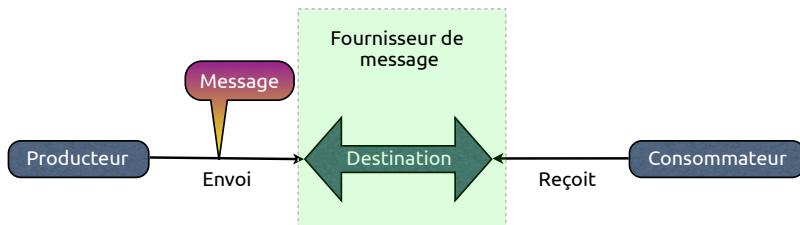
221

# Message

- Communication asynchrone
- Faiblement couplée
- Basé sur MOM

222

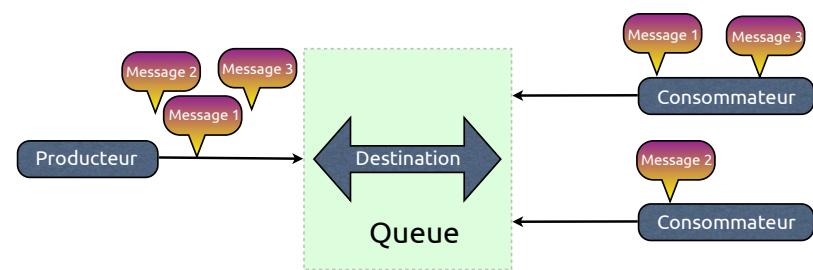
# MOM



- 2 stratégies
  - Point-To-Point (PTP)
  - Publish-Subscribe (pub-sub)
- ActiveMQ, SonicMQ, IBM Websphere MQ, etc...

223

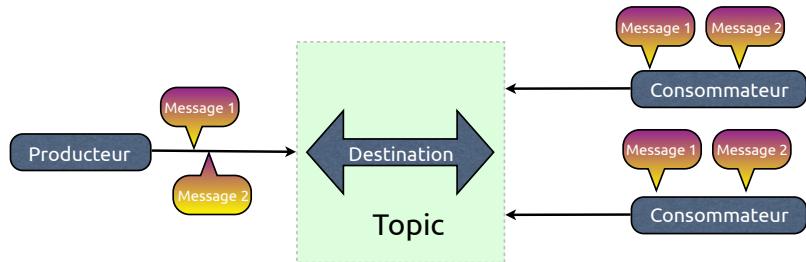
# Point-To-Point



- Pas de garantie d'ordre de délivrance
- Si il y a plusieurs récepteurs, le choix du récepteur est aléatoire

224

## Publish-Subscribe



- Timer entre Publisher & Subscriber sont liés
- Si un abonné est inactif au delà d'une période donnée, il ne recevra pas le(s) message(s)

225

## Request-Reply

- permet d'avoir un accusé réception des Consommateurs
- S'ajoute au dessus des deux précédents
- Pour permettre la réponse, ajout d'informations dans le message comme un identifiant unique et une Queue de destination pour la réponse en PTP

226

## Emetteur : exemple

```
@Resource(name = "jms/QueueConnectionFactory")
private ConnectionFactory connectionFactory;
@Resource(name="jms/MessageQueue")
private Destination destination;
public void sendMessage() {
    try {
        Connection connection = connectionFactory.createConnection();
        Session session = connection.createSession(true, Session.AUTO_ACKNOWLEDGE);
        MessageProducer messageProducer = session.createProducer(destination);
        textMessage.setJMSPriority(1);
        textMessage.setStringProperty("key", "value");
        TextMessage textMessage = session.createTextMessage();
        textMessage.setText("Hello from Blois's university");
        messageProducer.send(textMessage);
    } catch (JMSException e) {
        // TRAITEMENT DES ERREURS
    }
}
```

227

## Interface Message

- Semblable à un mail composé de trois parties
  - Headers : couples standards nom-valeur (JMSCorrelationID, JMSReplyTo, JMSMessageID, JMSTimeStamp...)
  - Properties : couples libre nom-valeur (primitives, String ou Objet)
  - Body : Contenu du message
- ByteMessage, MapMessage, StreamMessage, TextMessage, ObjectMessage

228

## Intérêt des MDBs

- Multithreading : pool de MDB dont une instance est extraite lors de l'arrivée d'un message
- Simplification du code du consumer

229

## Receveur (interface)

```
@MessageDriven(name = "receiverMDB"
, activationConfig = {
    @ActivationConfigProperty(
        propertyName = "DestinationType", propertyValue = "javax.jms.Queue"
    ),
    @ActivationConfigProperty(
        propertyName = "DestinationName", propertyValue = "jms/Queue"
    )
}
)
public class Receiver implements MessageListener {
    @Resource
    private MessageDrivenContext messageDrivenContext;

    @Override
    public void onMessage(Message message) {
        if (message instanceof TextMessage) {
            TextMessage textMessage = (TextMessage) message;
            // do some stuff with message
        }
    }
}
```

231

## Règles de programmation

- doit implémenter une interface MessageListener soit directement (implements) soit indirectement (annotation)
- doit être une classe concrète (ni final, ni abstract)
- doit être un POJO et pas sous classe de MDB
- doit être public
- Constructeur sans argument
- Ne pas lever de RuntimeException (termine l'instance du MDB)

230

## Receveur (annotation)

```
@MessageDriven(name = "receiverMDB"
, messageListenerInterface = javax.jms.MessageListener.class
, activationConfig = {
    @ActivationConfigProperty(
        propertyName = "DestinationType", propertyValue = "javax.jms.Queue"
    ),
    @ActivationConfigProperty(
        propertyName = "DestinationName", propertyValue = "jms/Queue"
    )
}
)
public class Receiver {
    @Resource
    private MessageDrivenContext messageDrivenContext;

    public void onMessage(Message message) {
        if (message instanceof TextMessage) {
            TextMessage textMessage = (TextMessage) message;
            // do some stuff with message
        }
    }
}
```

232

## @ MessageDriven

```
Target({TYPE}) @Retention(RUNTIME)
public @interface MessageDriven {
    String name() default DE;
    Class messageListenerInterface() default Object.class;
    ActivationConfigProperty[] activationConfig() default {};
    String mappedName();
    String description();
}
```

- name : nom du MDB dans l'arbre JNDI
- messageListenerInterface : type de message listener implémenté (sinon, il faut utiliser l'interface)
- activationConfig : propriétés de configuration

233

## Propriété acknowledgeMode

- Confirmation du container d'EJB au serveur JMS que le message a été bien reçu.
- AUTO\_ACKNOWLEDGE : confirmation dès réception du message (mode par défaut)
- DUPS\_OK\_ACKNOWLEDGE : la confirmation peut être envoyée en différé. Le MDB doit être capable de gérer les doublons.

235

## @ActivationConfigProperty

```
public @interface ActivationConfigProperty {
    String propertyName();
    String propertyValue();
}
```

- Les plus communs sont :
  - destinationType : Queue ou Topic
  - connectionFactoryJndiName
  - destinationName
  - acknowledgMode
  - messageSelector
  - subscriptionDurability

234

## Propriété subscriptionDurability

- Pour les MDB écoutant des Topics, on précise la durée de la souscription, c'est à dire si le MOM doit conserver une copie du message en l'absence du Consumer
  - Durable : conservation des messages
  - NonDurable : non conservation des messages (valeur par défaut)

236

## Propriété messageSelector

permet de déclarer un filtre pour le MDB avec une syntaxe proche du WHERE du SQL sur les Headers et Properties du message

Type	Description	Exemple
Littéral	String, numerique ou booléen	Chaine, 100,TRUE
Identifiant	message property ou header name	RECIPIENT, JMSTimestamp, Fragile, ...
<sup>8</sup> Whitespace <sup>8</sup>	cf. JLS (space, tab, form feed, line terminator)	
Opérateurs de comparaison	>, >= , = , < , <= , <>	RECIPIENT= <sup>8</sup> MonMDB <sup>8</sup> NumOfBids>=100
Opérateurs logique	NOT,AND, OR	Condition1 AND Condition2
Comparaison à NULL	IS [NOT] NULL	FirstName IS NOT NULL
Comparaison à TRUE/FALSE	IS [NOT] TRUE, IS [NOT] FALSE	Fragile IS TRUE

237

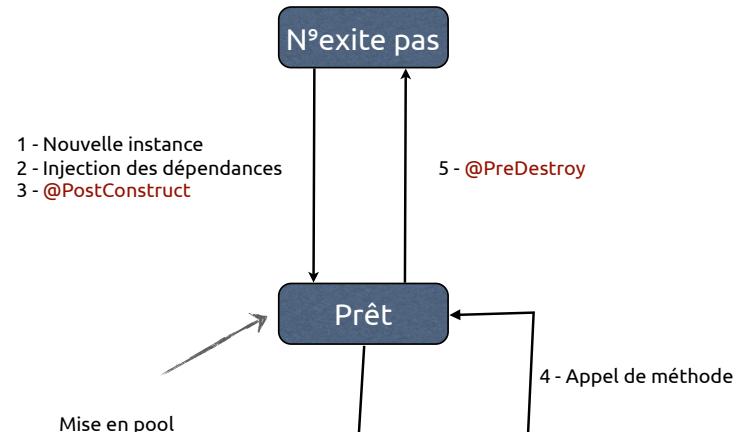
## MDB producteur (1)

```
@MessageDriven(name = "receiverMDB"
    , activationConfig = {@ActivationConfigProperty(propertyName = "messageSelector", propertyValue
    = "newItem > 10")})
public class Receiver implements MessageListener {
    @Resource
    private MessageDrivenContext messageDrivenContext;
    @Resource(name = "jms/QueueConnectionFactory")
    private ConnectionFactory connectionFactory;
    @Resource(name="jms/MessageQueue")
    private Destination destination;
    private Connection connection;
    @Override
    public void onMessage(Message message) {
        if (message instanceof TextMessage) {
            TextMessage textMessage = (TextMessage) message;
            printItemList();
        }
    }
    @PostConstruct
    private void initialization() throws JMSException {
        connection = connectionFactory.createConnection();
    }
    @PreDestroy
    private void tearDown() throws JMSException {
        connection.close();
    }
}
```

239

## Cycle de vie

Semblable au Stateless



238

## MDB producteur (2)

```
public class Receiver implements MessageListener {
    .....
    private void printItemList() throws JMSException {
        Session session = connection.createSession();
        MessageProducer messageProducer = session.createProducer(destination);
        TextMessage textMessage = session.createTextMessage();
        textMessage.setText("Print last items");
        messageProducer.send(textMessage);
        session.close();
    }
}
```

240

## MDBs Recommandations

- Choisir avec attention le modèle de message (PTP ou Pub-Sub)
- Découpler la logique du traitement dans une autre méthode que onMessage voir dans un Session Bean
- Choisir entre multiplier les destinations ou utiliser les filtres
- Choisir le type de message (XML permet le découplage mais augmente la charge)
- Attention aux messages empoisonnés (les MOM permettent de compter les <sup>8</sup>redelivery<sup>8</sup> et basculer sur une <sup>8</sup>dead message queue<sup>8</sup>)
- Dimensionner correctement le pool de MDBs

241

## Quelques définitions

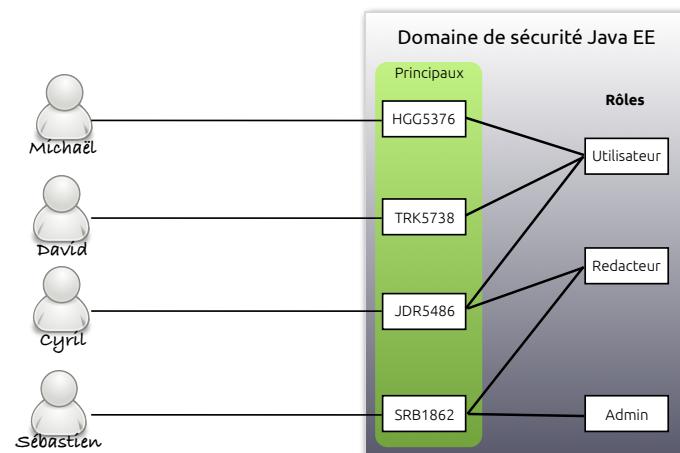
- Authentication : Vérification de l'identité de l'utilisateur
- Authorization : détermination si un utilisateur est autorisé à accéder à une ressource ou méthode
- User : personne reconnue par Authentication et associée à un Principal
- Groupe : regroupement logique de User au niveau du serveur
- Role : regroupement logique de User au niveau de l'application (peut être équivalent des groupes)
- Realm : scope sur lequel s'applique un politique de sécurité
- JAAS : Java Authentication and Authorization Service

243

## EJB & Sécurité

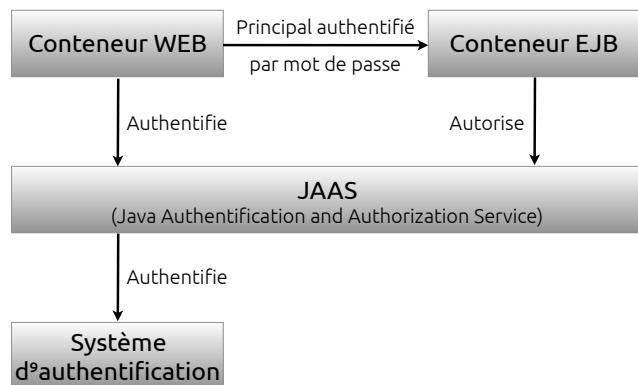
242

## Principal & Rôle



244

## Principe de fonctionnement



245

## Support de la sécurité

- Les EJB gèrent les autorisations
- Les clients gèrent l'authentification
  - Principal et rôle transmis aux EJB
- Sécurité déclarative
- Sécurité programmatique

246

## Sécurité déclarative

```
@Stateless  
@RolesAllowed({"admin", "writer", "user"})  
public class ActionEJB {  
  
    @RolesAllowed("admin")  
    public void restrictedAction() {  
        // some stuff  
    }  
  
    @PermitAll  
    public void action() {  
        // some stuff  
    }  
}
```

247

## Annotation de sécurité

Annotation	Bean	Méthode	Description
@PermitAll			Le bean ou la méthode est accessible à tous les rôles déclarés
@DenyAll			Aucun rôle n'est autorisé à exécuter
@RolesAllowed			Donne la liste des rôles autorisés
@DeclaredRoles			Définition des rôles de sécurité
@RunAs			Affecte temporairement un nouveau rôle au principal

248

## Sécurité programmatique

```
@Stateless  
public class ActionEJB {  
  
    @Resource  
    private SessionContext sessionContext;  
  
    public void restrictedAction() {  
        if (sessionContext.isCallerInRole("admin")) {  
            // do some stuff  
        } else {  
            throw new SecurityException("Only admin can do that");  
        }  
    }  
  
    public void action() {  
        if (sessionContext.isCallerInRole("writer")) {  
            // do some writer stuff  
        } else if (sessionContext.getCallerPrincipal().getName().equals("Cyril")) {  
            // do some stuff for Cyril  
        }  
        // do global stuff  
    }  
  
}
```

249

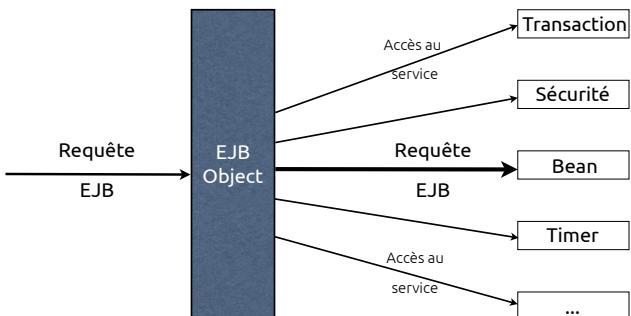
## EJB “forward”

EJB Context  
Managed bean  
CDI et @Resource  
Intercepteur

250

## EJB en coulisse

Au déploiement, le container génère un proxy appelé “EJB Object” qui accède à toutes les fonctionnalités du container



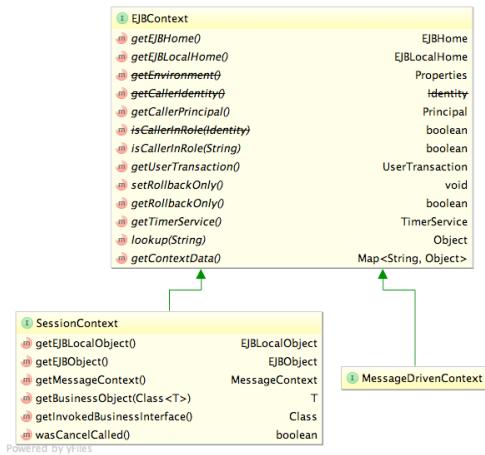
251

## EJB Context

```
public interface EJBContext {  
    EJBHome getEJBHome() throws IllegalStateException;  
    EJBLocalHome getEJBLocalHome() throws IllegalStateException;  
    Principal getCallerPrincipal() throws IllegalStateException;  
    boolean isCallerInRole(String s) throws IllegalStateException;  
    UserTransaction getUserTransaction() throws IllegalStateException;  
    void setRollbackOnly() throws IllegalStateException;  
    boolean getRollbackOnly() throws IllegalStateException;  
    TimerService getTimerService() throws IllegalStateException;  
    Object lookup(java.lang.String s) throws IllegalArgumentException;  
    Map<String, Object> getContextData();  
}
```

252

## EJB Context



253

## EJB Context - Utilisation

```
@Stateless  
public class MonSessionBean implement MonSession{  
    @Resource SessionContext sessionContext;  
    ..... }  
  
@MessageDriven  
public class MonMDB implements MessageListener {  
    @Resource MessageDrivenContext messageDrivenContext;  
    ..... }
```

254

## Managed bean

- Les “bean managé“ sont des POJO gérés par le container
- Modèle léger de composant
- Support d'un petit ensemble de services
  - Injection (@Resource, @Inject)
  - Contrôle de cycle de vie (@PostConstruct & @PreDestroy)
  - Intercepteur (@Interceptor, @AroundInvoke)

255

## Managed bean Vs EJB

Services	EJB	Managed bean
Injection de dépendance		
Intercepteur		
Cycle de vie		
Client distant		
Gestion d'état et pool		
Message		
Transaction & sécurité		
Concurrence		
Invocation Asynchrone		

256