Introduction to JPA

Fabio Falcinelli
Several years experience in active enterprise development

I love to design and develop web and standalone applications using
- Python
- Java
- C
- JavaScript

No troll wars...
- I mix languages in different phases of development
  - E.g. Java for the service, python for testing...
- Each tool has its own advantages
The course requires basic concepts and understanding about

- JAVA programming
- SQL expressions
- OOP patterns
- UML / ER representations
- Design patterns
Topics

- Overview of the JPA (Java Persistence API)
- Introducing the Auction Application
  - We will skip this one in favor to practice in classroom
  - Setting up a development environment with Eclipse IDE, JBoss and MySQL
  - Full working example of E-Commerce application with the Netbeans IDE
- Java Persistence API Entities
- Understanding the Entity Manager
- Modeling Entity Relationships
- Entity Inheritance and Object-Relational Mapping
- Persisting Enums and Collections
- Introduction to Querying
  - Using the JPA Query Language
  - Using the JPA Criteria API
- Advanced JPA concepts
Overview of the JPA
Designing a database driven app

- Usually, a database-driven app is developed starting by
  - An already existent database (BottomUp)
  - A Domain Model
  - An Entity/Relationship model
What is JPA?

- The Java Persistence API provides an **object/relational mapping facility** for managing relational data in Java applications.

- JPA is part of **EJB 3.0 specification** (**JEE >=5**)
  - But it is available for **JSE** too.

- It is a framework to solve the problem of **“object relational impedance mismatch”**.

- JPA does not free you from relational database understanding.
How can I represent my domain model on a relational database?

- Is-a relations
- Has-a relations
- State/Behaviour

JPA offers a full featured ORM (Object-Relational Mapper)
- Transparent persistence (POJO/JavaBeans)
- Persistent/Transient instances
- Automatic dirty checking
- Lazy fetching
- Outer Join fetching
- Runtime SQL generation
- Inheritance mapping strategies
Why I should use JPA instead of other ORM tools like Hibernate?

- Hibernate is a JPA implementation
- JPA has collected the best ideas from existing persistence technologies like
  - Hibernate
  - TopLink
  - JDO
- JPA is a standardized specification that helps you build a persistence layer that is independent of any particular **persistence provider**
N-Tier architecture

- Typically, an enterprise application is made up of layers (or tiers)
  - Specific responsibilities
  - Decoupling

- The most common setup is a three-tier application
  - Presentation
    - Responsible for displaying data only
    - This is the part the user interacts with
  - Business Logic
    - Provides services and algorithm
  - Persistence
    - Data representation and store/retrieval procedures
- It resembles very strictly the MVC pattern...
MVC pattern

Model
- Encapsulates application state
- Responds to state queries
- Exposes application functionality
- Notifies views of changes

View
- Renders the models
- Requests updates from models
- Sends user gestures to controller
- Allows controller to select view

Controller
- Defines application behavior
- Maps user actions to model updates
- Selects view for response
- One for each functionality

= Method Invocations
= Events
JPA in a MVC example
JPA key concepts

- **Entity**
  - A **domain object** for which it is important to **persist** the state

- **Entity Manager Interface**
  - An object implementing this interface performs the **real operations** to persist, delete, find entities **in/from the data storage**

- **Entity Manager Factory**
  - Creates the object implementing the entity manager interface
  - **Configuration** parameters are contained in a **persistence unit**

- **Persistence Context**
  - **Set of entities managed** by an entity manager
Questions?
Day One - II

Java Persistence API
Entities
An **entity** is a **lightweight persistence domain object**
- An entity typically represents a table
- An entity instance is a record into that table

The **persistent state** of an **entity** is represented through either
- persistent fields
- persistent properties
Entity requirements

- Must be annotated with the `javax.persistence.Entity` annotation
- Must have a `public` or `protected`, no-arg constructor
- Must *NOT* be declared `final`
  - class itself
  - methods
    - unless declared `transient` (or `@Transient` annotation)
  - persistent instance variables
    - unless declared `transient` (or `@Transient` annotation)
- Must implement the `Serializable` interface if an entity instance have to be passed by value as a detached object
- May extend both entity and non-entity classes, and vice versa
- Must declare `private`, `protected`, or `package` persistence attributes
  - Can be accessed directly only by the entity class’s methods
**JavaBean class**

```java
public class Vehicle {

    private int id;
    private String name;
    private long mileage;

    public Vehicle() {}
    public Vehicle(int id) {
        this.id = id;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    // ...
}
```

**Entity class**

```java
@Entity
public class Vehicle {

    @Id
    private int id;
    private String name;
    private long mileage;

    public Vehicle() {}
    public Vehicle(int id) {
        this.id = id;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    // ...
}
```
Entities may use
- persistent fields
- persistent properties
- a combination of both

Persistence fields
- the mapping annotations are applied to the entity’s instance variables

Persistence properties
- the mapping annotations are applied to the entity’s getter methods for JavaBeans-style properties
Allowed types for fields/properties

- Java primitive types
  - `java.lang.String`
- Other serializable types, including:
  - Wrappers of Java primitive types
    - `java.math.BigInteger`
    - `java.math.BigDecimal`
    - `java.util.Date`
    - `java.util.Calendar`
    - `java.sql.Date`
    - `java.sql.Time`
    - `java.sql.TimeStamp`
  - User-defined serializable types
  - `byte[]`
  - `Byte[]`
  - `char[]`
  - `Character[]`
  - Enumerated types
  - Other entities and/or collections of entities
  - Embeddable classes
- The Persistence runtime accesses entity-class instance variables **directly**

- All fields not annotated as `javax.persistence.Transient` or not marked as `Java transient` will be persisted to the data store

- The object/relational **mapping annotations** must be **applied** to the **instance variables**
Field access

- getter/setter are optional
- must be either
  - protected
  - package
  - private

```java
@Entity
public class Vehicle {

@Id private int id;
private String name;
private long mileage;

...}
```
- JavaBeans method conventions apply
- For each persistent property there is
  - a **getter** method (e.g. `getName()`)
  - a **setter** method (e.g. `setName(String name)`)  
  - If the property is a Boolean, you may use `isPropertyName` instead of `getPropertyName`

- The object/relational **mapping annotations** for persistent properties must be **applied** to the **getter** methods
- Mapping annotations *CANNOT* be applied to fields or properties annotated `@Transient` or marked `transient`
Persistence property example

- Property access
  - getter/setter are required
  - getter determines the type of property
  - getter should be declared
    - public
    - protected

```java
@Entity
public class Vehicle {

  private int id;
  private String name;
  private long mileage;

  @Id public int getId() {
    return id;
  }
  public void setId(int id) {
    this.id = id;
  }
  public String getName() {
    return name;
  }

  ...
}
```
- **Collection-valued** persistent fields and properties must use the supported Java collection interfaces
  - java.util.Collection
  - java.util.Set
  - java.util.List
  - java.util.Map

- **Example**

  ```java
  Set<Owner> getOwners() { ... }
  void setOwners(Set<Owner> owners) { ... }
  ```
Use the `javax.persistence.ElementCollection` annotation on the field or property is
- A collection of basic types
- A collection of embeddable classes

It allows 2 parameters to be set
- `targetClass`
- `fetchType`

```java
@Entity
public class Developer {
    //...
    @ElementCollection(fetch=FetchType.EAGER)
    protected Set<String> nicknames = new HashSet();
    //...
}
```
Every entity must have a unique object identifier (the **primary key**)

An entity may have either

- A **simple** primary key
  - use the `javax.persistence.Id` annotation on property or field
  - We will refer to this as “**primary key**” from now on
- A **composite** primary key
  - use the `javax.persistence.EmbeddedId` and `javax.persistence.IdClass` annotations
  - We will see this in the **advanced concepts**...
@Id requirements

- Java primitive types
- Java primitive wrapper types
  - java.lang.String
  - java.util.Date
  - java.sql.Date
  - java.math.BigDecimal
  - java.math.BigInteger
- Floating-point
  - Although allowed should never be used for primary keys
  - Only integral types will be fully portable
- Id cannot be changed once assigned
  - JPA behavior is undefined if this occurs
- Primary keys (aka ids) can be created automatically by the persistence provider
- Use the @GeneratedValue annotation to set the strategy
  - IDENTITY
    - The db assigns the id value
    - Id accessibility is not guaranteed before an insert operation committed
  - SEQUENCE
    - Works with databases supporting sequences
    - @SequenceGenerator can be defined
  - TABLE
    - Using a generator table (portable)
    - @TableGenerator can be defined
  - AUTO (default)
    - Persistence provider choose (rapid development)
JPA assumes
- Table name is equal to entity name
- Column name is equal to field/property name

@Table is used to override table properties such as name, schema...

@Column is used to override mainly the column name

```java
@Entity
@Table(name="VEH_TABLE")
public class Vehicle {
    @Id
    @Column(name="VEH_ID")
    private int id;

    private String name;
    @Column(name="VEH_MIL")
    private long mileage;
    // ...
}
```
Questions?
Day One - III

Understanding the Entity Manager
Entity Manager

- Entities are managed by an EntityManager
  - An instance of javax.persistence.EntityManager

- Each EntityManager instance is associated with a **persistence context**
  - The persistence context is a set of managed entity instances that exist in a particular data store
  - A **persistence context** defines the **scope** under which particular entity **instances** are created, persisted, and removed

- The **EntityManager** interface defines the **methods** that are used to interact with the **persistence context**
You obtain an EntityManager instance from an EntityManagerFactory.

There should be one entity manager factory per persistence unit.

Thread-safe

```java
private static final String PERSISTENCE_UNIT = "VehicleService";

private EntityManagerFactory createFactory() {
    return Persistence.createEntityManagerFactory(PERSISTENCE_UNIT);
}
```
The most basic operation is to retrieve an entity from the data storage.

The `find` method is used to look up entities in the data store by the entity’s primary key.

When retrieving an entity, it becomes part of the EntityManager's persistence context. It's state turn to **managed**

```java
@PersistenceContext
EntityManager em;

public void enterOrder(int custID, Order newOrder) {
    Customer cust = em.find(Customer.class, custID);
    cust.getOrders().add(newOrder);
    newOrder.setCustomer(cust);
}
```
Entity states

- **New**
  - entity instances have no persistent identity and are not yet associated with a persistence context

- **Managed**
  - entity instances have a persistent identity and are associated with a persistence context

- **Detached**
  - entity instances have a persistent identity and are not currently associated with a persistence context

- **Removed**
  - entity instances have a persistent identity, are associated with a persistent context, and are scheduled for removal from the data store
Each arrow is an entity manager method (except `new`), each balloon is an entity state.
- Entity’s data is stored to the database when the transaction associated with the persist operation is completed (commit).
- New entity instances become managed:
  - Invoking the persist method
  - By a cascading persist operation
    - `cascade=PERSIST, cascade=ALL`
- If `persist` is called on:
  - An already managed entity instance:
    - The persist operation is ignored
    - The persist operation will eventually cascade to related entities
  - A removed entity instance, then:
    - The entity becomes managed
  - A detached instance, then either:
    - `persist` will throw an `IllegalArgumentException`
    - The transaction commit will fail
Removing entity instances

- Entity’s data will be removed from the data store when the transaction is completed or as a result of the `flush` operation.

- Managed entity instances are removed by:
  - Invoking the remove method.
  - By a cascading remove operation:
    - `cascade=REMOVE, cascade=ALL`.

- If the `remove` method is invoked:
  - On a new entity:
    - The remove operation is ignored.
    - Remove will eventually cascade to related entities.
  - On a detached entity, either:
    - Remove will throw an `IllegalArgumentException`.
    - The transaction commit will fail.
  - On an already removed entity:
    - Remove will be ignored.
To force synchronization of the managed entity to the data store, invoke the `flush` method of the `EntityManager` instance

If the entity is related to another entity and the relationship annotation has the `cascade` element set to `PERSIST` or `ALL`, the related entity’s data will be synchronized with the data store when `flush` is called

If the entity is removed, calling `flush` will remove the entity data from the data store
Entities can be detached by

- Calling `detach` method on a given entity
- Closing the entity manager or using the `clear` method on the entity manager detaches all entities

If the data of the detached entity must be persisted

- Call `persist`, the entity will be managed again
- Call `merge` to store the data
  - In this case, the entity remains in detached state
- A set of all entities managed by the EntityManager
- Represents the data contained within a single data store

```xml
<persistence>
  <persistence-unit name="VehicleService"
    transaction-type="RESOURCE_LOCAL">
    <class>it.garage.Vehicle</class>
    <properties>
      <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
      <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/GarageDB;create=true"/>
      <property name="javax.persistence.jdbc.user" value="admin"/>
      <property name="javax.persistence.jdbc.password" value="pippo123"/>
    </properties>
  </persistence-unit>
</persistence>
```
The XML file is placed in the META-INF folder

- `persistence.xml` can have multiple persistence units
  - Each unit can be used by different JPA vendor or can be used to persist to different databases

- The vendor-specific persistence provider name is specified in the `<provider>` tag

- The entity class names are specified in the `<class>` tag
  - Each entity must be listed separately, the * is not supported

- The database connection properties can be specified within the `<properties>` tag
  - Note that the property name will differ for each vendor
/*VehicleService is the persistence unit configuration that
determines things as the connection parameters that will be used
when connecting to the database */
EntityManagerFactory emf = Persistence.createEntityManagerFactory("VehicleService");

//With the factory create an entity manager
EntityManager em = emf.createEntityManager();

//Create an entity vehicle
Vehicle v = new Vehicle(83);

//Call the entity manager to persist the entity just created
try{
    em.persist(v);
    //From now on, the entity is managed by the entity manager
    //within its persistence context
    ...
}catch(PersistenceException e){
    e.printStackTrace();
}
/* Assuming to have the entity manager as before retrieve the vehicle from the database */
Vehicle v = v.find(Vehicle.class, 83);

// If the entity is not found, find method return null
if (v != null) {
    /* We can change its mileage
    by calling this method, changes are reflecting
directly to the persistence context, and
then the database without explicitly using
the entity manager */
    v.setMileage(v.getMileage() + 5000);

    /* Then wreck it! :-) */
    em.remove(v);
    // Here the entity is in removed state
} else {
    // This would throw an IllegalArgumentException X-(
    em.remove(v);
}
A **Java EE container** manages the **lifecycle** of container-managed entity managers

An EntityManager instance’s persistence context is automatically propagated by the container to all application components that use the EntityManager instance within a **single JTA transaction**

To obtain an EntityManager instance, **inject** the entity manager into the application component with the `javax.persistence.PersistenceContext` **annotation**

```java
...  
@PersistenceContext
EntityManager em;
... 
```
The persistence context is **NOT** propagated to application components, and the **lifecycle** of **EntityManager instances** is **managed** by the application.

Applications create EntityManager instances in this case by using the `createEntityManager` method of `javax.persistence.EntityManagerFactory`.

To obtain an EntityManager instance, you first must obtain an EntityManagerFactory instance by injecting it into the application component by means of the `javax.persistence.PersistenceUnit` annotation.
/**inject the factory by the persistence unit annotation*/

@PersistenceUnit
EntityManagerFactory emf;
EntityManager em;
@Resource
UserTransaction utx;

// ...
/**request an entity manager instance to the factory*/
em = emf.createEntityManager();
try {
    /**start a transaction*/
    utx.begin();
    /**call persistence methods on entities*/
    em.persist(SomeEntity);
    em.merge(AnotherEntity);
    em.remove(ThirdEntity);
    /**commit the transaction*/
    utx.commit();
} catch (Exception e) {
    /**if a error occurs, rollback*/
    utx.rollback();
}
Questions?
Overview of the Java Persistence API
- Describe the basics of **Object Relational Mapping** (ORM)
- Define the key concepts of the Java Persistence API (**entity**, **entity manager**, and **persistence unit**)

Java Persistence API Entities
- Describe the difference between **objects** and **entities**
- Describe the difference between **persistent fields** and **properties**
- Identify and use common Java Persistence API annotations, such as **@Entity**, **@Id**, **@Table**, and **@Column**

Understanding the **Entity Manager**
- Describe the relationship between an **entity manager**, a **persistence context**, and a **persistence unit**
- Describe the difference between a **container-managed** entity manager and an **application-managed** entity manager
- Describe the **entity life cycle**
References

- **The Java EE 6 Tutorial**
- **The NetBeans E-commerce Tutorial**
  - http://netbeans.org/kb/docs/javaee/ecommerce/intro.html