JPA with Hibernate 3.0

Java Persistence API

Lesson Objectives

After completing this lesson, participants will be able to understand:

- Java Persistence API

- Working with JPA

- Managing entities using EntityManager

This lesson is startup for setting up JPA in our application and explains how to perform basic operations on entities using JPA interfaces/classes.

JPA Overview

Entity Classes

EntityManager

▪ Persistence Context

EntityManagerFactory

EntityTransaction
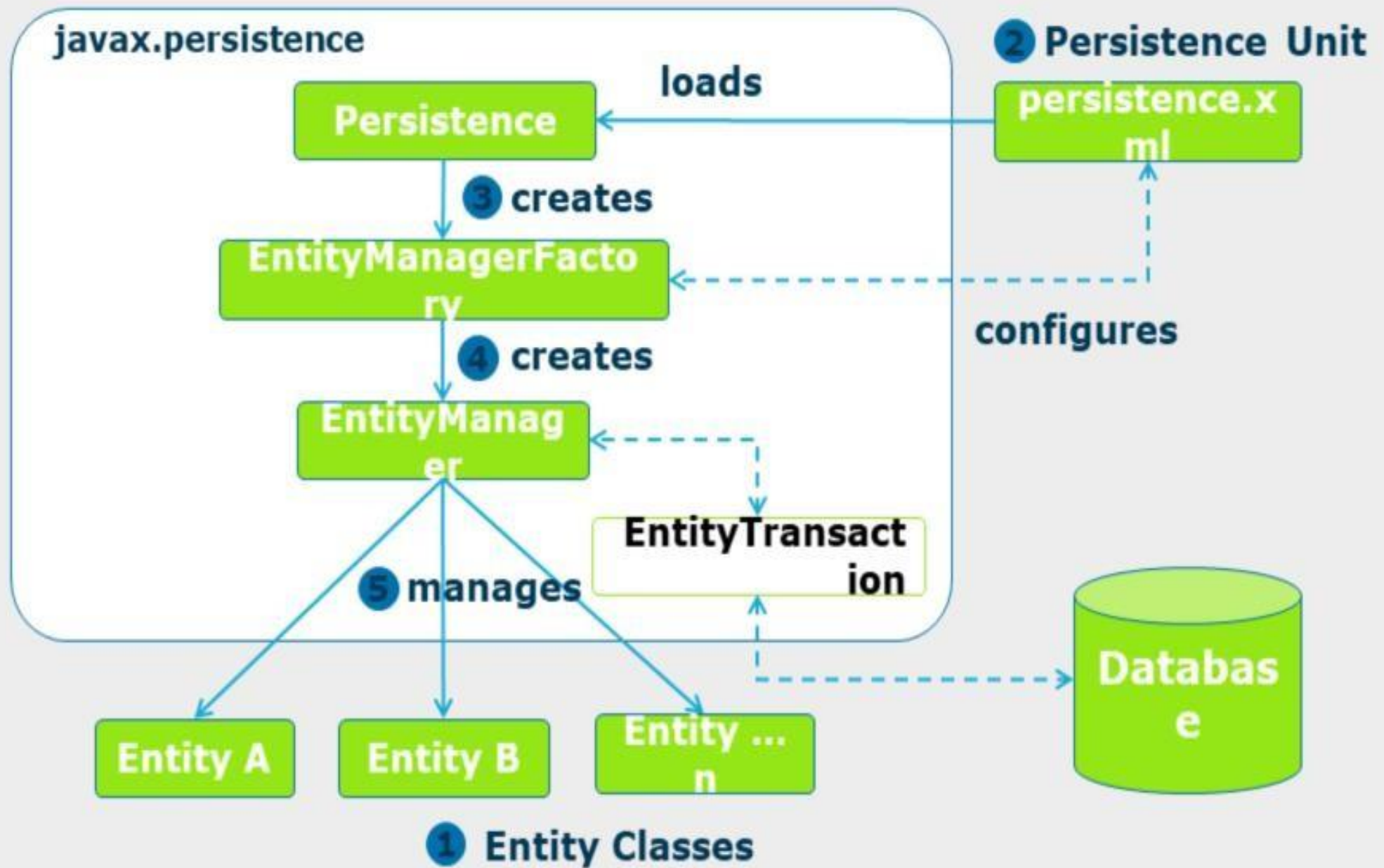
Persistence Unit

▪ persistence.xml

Java Persistence Query Language (JPQL)

▪ Query

The shows important components of JPA that each application uses to communicate with the database.

This lesson gives an introduction about what are those components, their functionality and how to configure them in our application.

Working with JPA

The slide shows typical JPA components interaction/working.

1.You normally start with a persistence strategy by identifying which classes need to be made entities.

2.Next step is to create configuration file (an XML document named persistence.xml) that contains the details about the relational database.

3.EntityManagerFactory is an factory based class responsible for creating EntityManager instance. It is obtained using Persistence class's createEntityManagerFactory static method.

4.EntityManagerFactory class designed to create EntityManager.

5.Once you have an EntityManager, you can start managing your entities. You can persist an entity, find one that matches a set of criteria, and so on. Each work of EntityManager with entities must be governed under EntityTransaction.

Let us discuss the each step in detail.

Creating Entity Classes

Java POJO classes can be made entity via either one of the following way:

- XML configuration (orm.xml)

- Annotations

A single xml file i.e. orm.xml is required to map entire set of entity classes within an application.

In case of Annotations, it should be marked in individual classes.

```
@Entity
public class Student implements Serializable {
    @Id
    private int studentId;
    private String name;
    //getters and setters
}
```

Requirements for Entity Classes:

The class must be annotated with the javax.persistence.Entity annotation.

The class must have a public or protected, no-argument constructor. The class may have other constructors.

The class must not be declared final. No methods or persistent instance
variables must be declared final.

Entities may extend both entity and non-entity classes, and non-entity classes may extend entity classes.

Persistent instance variables must be declared private, protected, or package- private and can be accessed directly only by the entity class's methods.
Clients must access the entity's state through accessor or business methods.

Note: As there are two ways to configure entities, either in XML (orm.xml) or with annotations, to keep contents **simple and manageable**, this course focuses **only on annotations** to configure entity classes.

Entity Annotations

Mandatory Annotations

- @Entity

- @Id

Few more Annotations

- @GeneratedValue

- @Table

- @Column

- @Transient

```
@Entity
@Table(name="student_masters")
public class Student implements Serializable {
    @Id
    @GeneratedValue(strategy=GenerationType.
AUTO)
    @Column(name = "stud_id")
    private int studentId;
    private String name;
```

The **@Entity** annotation marks this class as an entity bean, so it must have a no-argument constructor that is visible with at least protected scope.

Each entity bean has to have a primary key, which you annotate on the class with the **@Id** annotation.

In some situation, few properties of an entity, do not need to be stored in the database. In this case, ORM do not take this property for all the Database operation. This can be done using **@Transient** annotation.

By default, the @Id annotation will automatically determine the most appropriate primary key generation strategy to use—you can override this by also applying the **@GeneratedValue** annotation. This takes a pair of attributes: **strategy and generator**

The **strategy** attribute must be a value from the **GeneratorType** enumeration, which defines four types of strategy constants.

1. **AUTO: (Default)** JPA decides which generator type to use, based on the database's support for primary key generation.

1. **IDENTITY:** The database is responsible for determining and assigning the next primary key.

3. **SEQUENCE:** Some databases support a SEQUENCE column type.

4. **TABLE:** This type keeps a separate table with the primary key values.

Note: Identity strategy depends upon database, for example, if database (MySQL) support AUTO INCREMENT column, then we can use IDENTITY to support this feature. For SEQUENCE and TABLE generator, see further notes.

Persistence Configuration

JPA persistence configuration is done with an XML file called "persistence.xml" which contains information about how to connect to the underlying database.

```xml
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
        http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
version="2.0">
<persistence-unit name="unit-name ">
        <provider> <!-- JPA provider name like hibernate-->
</provider>
        <properties>  <!-- Database properties --></properties>
    </persistence-unit>
</persistence>
```

To connect with database, you need to set various properties regarding driver class,

 user name and password. This configuration is done with an XML file named persistence.xml.
Elements in persistence.xml:

The <persistence> is the root element of persistence.xml file. A persistence unit defines all the entity classes that need to be managed and the JDBC details to connect to an underlying relational database.

<persistence-unit> : It has the name attribute specifies a name that can be referenced from your Java code. The transaction-type attribute informs ORM about transaction management. It may take values like:
RESOURCE_LOCAL: Application will handle transaction management. i.e. creating, starting and closing of transactions.
JTA: JEE server Container will take care for transaction management.

<provider>: Specifies the fully-qualified name of the JMS provider class. E.g. hibernate.

<property>: Minimum four properties must be nested using <property> element . needed. These properties specify the JDBC URL, JDBC username, JDBC password, and driver.

<class> : Each class element specifies a fully-qualified name of an entity class. This approach is used to inform which classes needs to be managed by JPA. i.e. Entity classes. There may be more than one class elements.

Note: This configuration file must be stored under META-INF directory of your application project.

Obtaining Entity Manager

An EntityManager is responsible for managing entities.

Below are the steps to create instance of EntityManager.

- Create EntityManagerFactory by using the Persistence class

- Call createEntityManager() on an EntityManagerFactory.

```
EntityManagerFactory emf =
Persistence.createEntityManagerFactory(name);
EntityManager em = emf.createEntityManager();
```

An EntityManager is responsible for managing entities. It is one of the most important types in the API.

You can get an EntityManagerFactory easily by using the Persistence class's createEntityManagerFactory() static method. It accept string parameter which is name of persistence unit defined in persistence.xml file.

Using the EntityManagerFactory class factory, you can create EntityManager
instances using createEntityManager() method.

Note: In your application when there is no use of
EntityManagerFactory or application shuts down then it is necessary to close the instance
of **EntityManagerFactory** . Once the **EntityManagerFactory** is closed, all its EntityManagers are also closed.

Working with Entity Manager

The EntityManager interface is providing the API for interacting with the Entity.

- Creates and removes persistent entity instances

- Finds entities by their primary key

- Allows for data querying

- Interacts with the persistence context

Following are some of the methods of EntityManager

Task	EntityManager method
Save new/detached entity	persist(entity-instance)
Update state of entity	merge(entity-instance)
Remove entity	remove(entity-instance)
Search for entity	find(class,idvalue)

The EntityManager interface defines the methods that are used to interact with the persistence context. The EntityManager API is used to create and remove persistent entity instances, to find persistent entities by primary key, and to query over persistent entities.

EntityManager important methods:

1. **persist(object)**: Persists the entity object
2. **find(class,primarykey)**: Retrieves a specific entity object
3. **remove(object)**: Removes an entity object
4. **refresh**: Refreshes the entity instances in the persistence context from the database
5. **contains**: Returns true if the entity instance is in the persistence context. This signifies that the entity instance is managed
6. **flush**: forces the synchronization of the database with entities in the persistence context
7. **clear**: Clears the entities from the persistence context
8. **evict(object):** Detaches an entity from the persistence context
9. **close():** Flush entity instances first, clears persistence context and nullify
   the entity manager

Persisting entity with Entity Manager

```java
public static void main(String[] args) {
    EntityManagerFactory factory =
            Persistence.createEntityManagerFactory("JPA-PU");
    EntityManager em = factory.createEntityManager();
    em.getTransaction().begin();
    Student student = new Student();
    student.setName("John");
    em.persist(student);
    em.getTransaction().commit(
    em.close();
    factory.close();
}
```

```java
@Entity
public class Student implements
Serializable {
    @Id
    private int studentId;
    private String name;
}
```

The above example shows how to persist entity instance using EntityManager.

void persist(java.lang.Object entity)

Persists an entity. The method throws an EntityExistsException if the entity already exists and a java.lang.IllegalArgumentException if the passed in object is not an entity.

Important: While managing instances of entities using EntityManager like saving, updating or removing, it is very much required to work in transaction. Therefore you need work with

EntityTransaction getTransaction()

method that returns resource-level EntityTransaction which can be used to
begin, **commit** or **rollback** transactions.

Note: You do not need an EntityTransaction for read-only operations. For
example, finding entity with EntityManager.find() method.

Demo

JPAStarter Demo

Entity CRUD with Entity Manager

```java
public Student getStudentById(int id) {
        Student student = entityManager.find(Student.class, id);
        return student;
}

public void addStudent(Student student) {
        entityManager.persist(student);
}

public void removeStudent(Student stude
        entityManager.remove(studer
}

public void updateStudent(Student studer
        entityManager.merge(student
}
```

```java
@Entity
public class Student impl....
{
        @Id
        private int studentId;
        private String name;
}
```

The above example shows how to perform CRUD
(create-read-update-delete) operations on entity
instance using EntityManager.

Please debug the demo, don't run it. While debugging the Client class, ensure to display state of table for each breakpoint.

4.2: Working with JPA

Demo

JPACrudOperations Demo

The above demos shows how to perform CRUD operations on entity in an layered architecture.

Lab

Lab 1

Summary

In this lesson, you have learned about:

- Setting up JPA in an application

- Configuring database with JPA

- Entity operations using EntityManager

Summary

Review Question

Question 1: Which of the following method/s is/are used to complete the transaction?

- Option 1: EntityManager.commit()

- Option 2: EntityManager.getTransaction().commit()

- Option 3: EntityManager.getTransaction().rollback()

- Option 4: All of above

Question 2: Persistence configuration file "persistence.xml" must be saved under META-INF folder.

- True/False