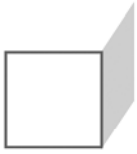
Chapter 4

Entity Managers



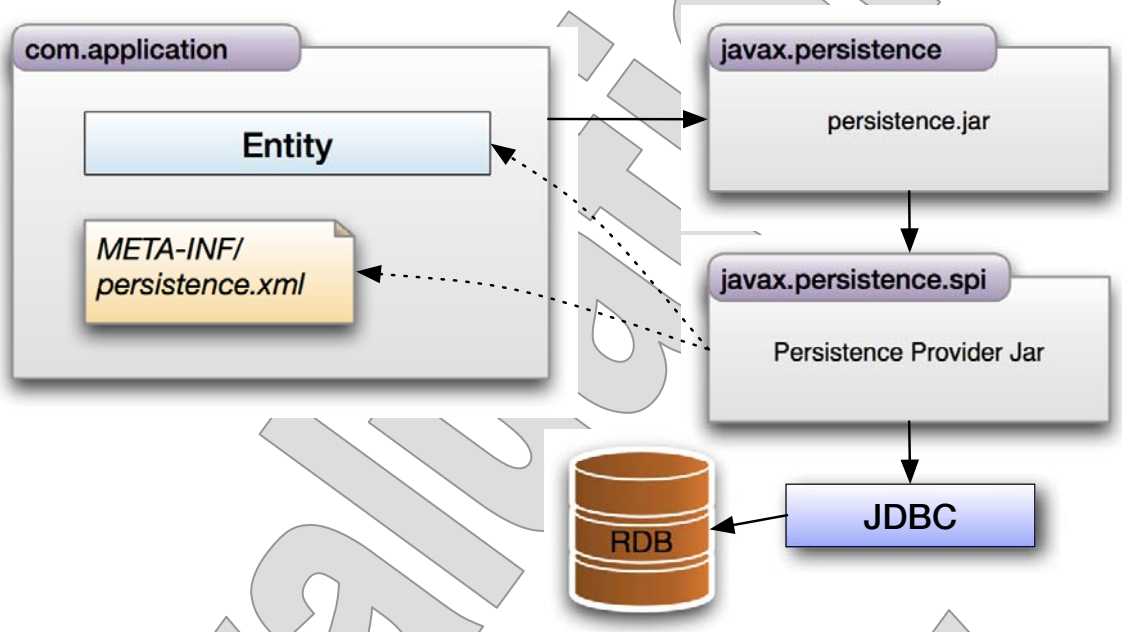
OBJECTIVES

After completing "," you will be able to:

- Write a **persistence.xml** file to declare persistence units and their elements.
- Describe the purpose of the **persistence context**.
- Create an entity manager factory, entity manager, and entity transaction.
- Use transactions, rollback, and entity state effectively.
- Use entity manager "CRUD" operations.
- Explain database synchronization, and when it takes place.
- Describe the details of detachment and merging during the entity lifecycle.
- Define and execute JPA queries.
- Pass parameters to queries.
- Explain why named queries are superior to dynamic queries.

Putting Entities to Work

- We are now going to use our annotated entities for **CRUD** (Create, Retrieve, Update, and Delete) operations.
- Let's examine the persistence system in more detail.



- **META-INF/persistence.xml** records a lot of the information we need in order to run our applications.
  - There can be more than one **persistence.xml** file in an application, but is common to have only one.
  - It contains the name or names of one or more **persistence units** that we will utilize in our application.
- The **persistence.xml** file can name the **persistence provider** to use in the application – or a default provider will be used.
- Depending on its type, the **persistence.xml** can go on to name the **JDBC driver** and **database** to be used.

The persistence.xml File

EXAMPLE

- In **Earthlings_Step5/src/META-INF**, we find the **persistence.xml** file, and possibly others as well; see the listing on the following page.
- The transaction type of this persistence unit is **RESOURCE_LOCAL**, which is typical for Java SE applications.
- The **persistence provider** is not specified, which leaves it to be detected at runtime.
  - In our case we assure that EclipseLink is the first JPA provider on the run-time class path.
- We could instead include a **<provider>** element whose value would be the class name of the desired **PersistenceProvider** implementation – such as ...

```
org.eclipse.persistence.jpa.PersistenceProvider
org.hibernate.ejb.HibernatePersistence
org.apache.openjpa
  .persistence.PersistenceProviderImpl
```

  - Notice that two such elements are prepared, inside an XML comment, in all of our **persistence.xml** files, so that you can easily force use of either EclipseLink or Hibernate in testing.

The persistence.xml File

EXAMPLE

- **The class file names for the entities are shown next, with one `<class>` element for each entity.**
  - An alternative is to list none of them; the following triple-negative phrase essentially tells the provider to scan the class path for JPA-annotated classes, including entities, embeddables, and converters:

```
<exclude-unlisted-classes>false
  </exclude-unlisted-classes>
```

- **We see several `<property>` elements where the name starts with `javax.persistence`; these are standard elements defined by the JPA specification.**
- **Vendor-specific `<property>` elements are often included as well, and we see the final property as an example.**
  - This element, when set to **FINE**, sets a logging level that will show generated SQL output, among other things.
  - There are similar versions for Hibernate and OpenJPA that we will see later.

The persistence.xml File

EXAMPLE

```
<persistence-unit
  name="EarthlingsPU"
  transaction-type="RESOURCE_LOCAL"
>
  <!-- Comment with specific <providers> -->

  <class>cc.hr.entity.Address</class>
  ...
  <class>cc.hr.entity.Project</class>

  <properties>
    <property
      name="javax.persistence.jdbc.url"
      value="jdbc:derby://localhost:1527/Capstone"
    />
    <property
      name="javax.persistence.jdbc.user"
      value="earthlings"
    />
    <property
      name="javax.persistence.jdbc.password"
      value="earthlings"
    />
    <property
      name="javax.persistence.jdbc.driver"
      value="org.apache.derby.jdbc.ClientDriver"
    />
    <property
      name="eclipselink.logging.level"
      value="WARNING"
    />
  </properties>
</persistence-unit>
```

The persistence.xml File

EXAMPLE

- **Or, if you've configured the labs for Oracle as described in Chapter 1, you'll see a few differences.**
  - If configured for Derby, you can still see the Oracle version of the file in **rdbms/oracle/META-INF**.
  - This version includes some placeholder tokens that get replaced with the content shown below when you re-configure for Oracle.
- **Naturally, a couple of the properties will need to be different:**

```
<property
  name="javax.persistence.jdbc.driver"
  value="oracle.jdbc.driver.OracleDriver"
/>
<property
  name="javax.persistence.jdbc.url"
  value="jdbc:oracle:thin:@localhost:1521:xe"
/>
```

  - Or a specific name or **IP address** may be seen instead of **localhost**, depending on your Oracle configuration as in Chapter 1.
- **There's also a reference to a second file:**

```
<mapping-file>META-INF/orm.xml</mapping-file>
```

The persistence.xml File

EXAMPLE

- **In that orm.xml file, you see an example of the ability to override mapping metadata as stated in source-code annotations.**
  - All of the entities in this project are defined to rely on Derby's **identity column** feature in order to generate new IDs.

```
@Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
private int id;
```

  - For Oracle, this won't work, and rather than replacing all the entity classes in order to control those annotations, we can **override** by stating different metadata for those specific spots:

```
<entity-mappings ... >

  <package>cc.hr.entity</package>

  <entity class="Location">
    <attributes>
    <id name="id">
      <generated-value strategy="SEQUENCE"
        generator="locationGen" />
      <sequence-generator
        name="locationGen"
        sequence-name="location_sequence"
        allocation-size="1"
      />
    </id>
    </attributes>
  </entity>
  ...
</entity-mappings>
```

JTA Persistence Units

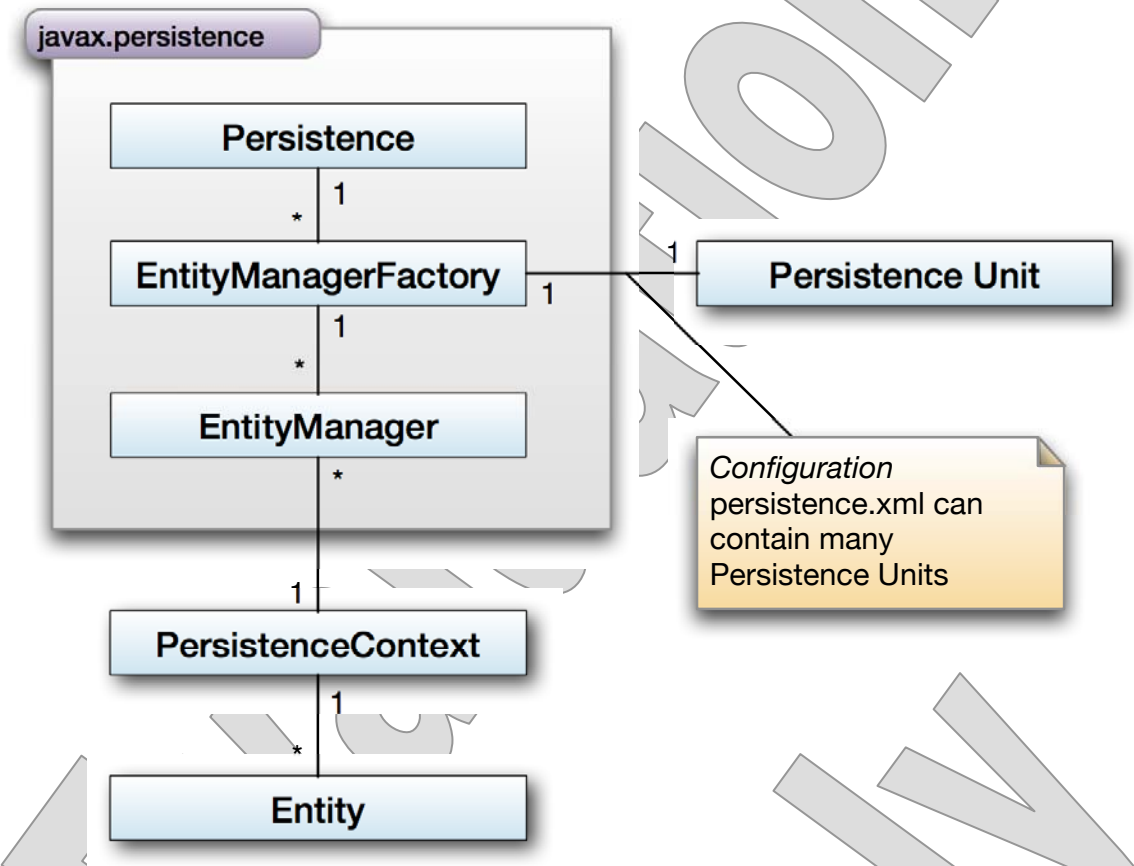
- A **JTA** (Java Transaction API) version of the **persistence.xml** file might look like this:

```
<persistence-unit
  name="EarthlingsPU"
  transaction-type="JTA"
>
  <jta-data-source>jdbc/EarthlingsDS
    </jta-data-source>
</persistence-unit>
```

- **This type of persistence unit is seen in the Java EE environment, where JTA data sources are used.**
  - We're telling the JPA provider that it can expect transactions to be **managed externally**, for example by an EJB or Spring container.
  - In turn, the provider will prohibit use of the **EntityTransaction** API made available by the entity manager for transaction control – saying, essentially, you can't implement them manually if they are already being managed automatically.
- **Also, in Java EE, the application server will automatically find the entity classes, so they do not need to be specified.**
  - That is, the default for **`<exclude-unlisted-classes>`** is **false** in a Java-EE container.
  - So you don't need to list the classes or state this directive as you might for Java SE; but note that even under Java SE most providers will find all local entities even if they are not explicitly listed.

Creating the Entity Manager

- Entity managers perform all the real work of CRUD operations, and are configured to read and write to the database.



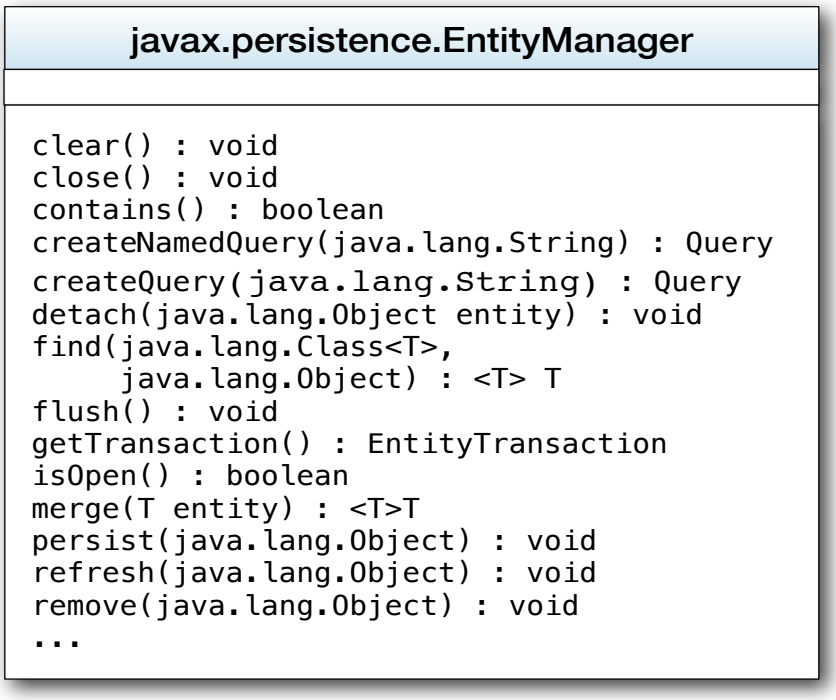
- The set of managed entity instances within the entity manager are known as the **persistence context**.
  - It is possible for a single persistence context to be associated with more than one entity manager, but we will not concern ourselves with this detail for now.
  - It is guaranteed that only **one instance** of an entity with the same persistent identity will exist in this persistence context at one time.

Creating the Entity Manager

- We use the **Persistence** class to create an **EntityManagerFactory**, according to the configuration of its associated persistence unit:

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

- There is a one-to-one correspondence between the persistence unit and its **EntityManagerFactory**.
- **EntityManagerFactory** then creates an **EntityManager**.



Showing Generated SQL

- **It is also possible to pass a Map of properties to the createEntityManagerFactory method.**
  - The property names are the same as those that can be used in **persistence.xml**.
  - Any properties defined in this map will **override** properties with matching names in the XML.
- **For example, we can ask the JPA provider to show generated SQL.**
  - The property names and values vary by provider.
  - It is fine to set a **superset** of those for multiple providers, as any properties not understood by a JPA provider will be ignored.
- **You might then trigger these property settings dynamically, based on a command-line argument or system property.**
- **A common setup for EclipseLink is:**

```
Map<String,String> properties = new HashMap<>();
properties.put
  ("eclipselink.logging.level.sql", "FINE");
properties.put
  ("eclipselink.parameters", "true");
emf = Persistence.createEntityManagerFactory
        (puName, properties);
```

  - Or, directly in **persistence.xml**:

```
<property name="eclipselink.logging.level.sql"
          value="FINE"/>
 <property name="eclipselink.logging.parameters"
           value="true"/>
```

Entity State

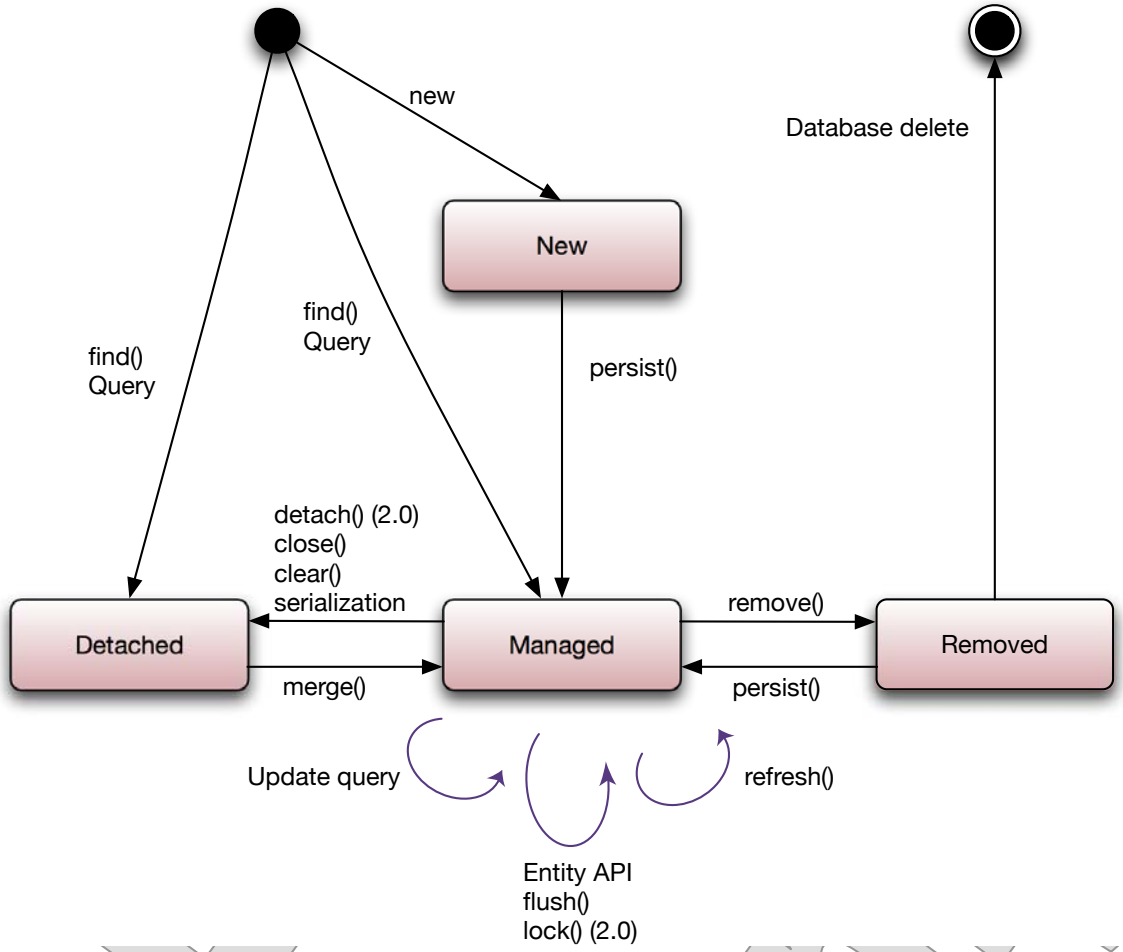
- **An instance of a JPA entity class can be in one of a few states, as defined by the specification and as pertains to their relationship to the entity manager.**
- **When an entity manager is aware of an instance, and is actively managing interactions with it, that entity is said to be in the managed state, and several promises are made:**
  - **There must be a row** in the corresponding database table with the primary key of that entity instance in memory.
  - There will only be **one managed instance** of that entity class with that primary key; the entity manager maintains an **entity cache**.
  - This guards against inconsistent data in memory that would eventually cause consistency problems in the database itself.
  - It also has a small performance advantage, because the entity manager can avoid repeated queries for the same entity.
  - The cache lives as long as the **persistence context** exists – which, depending on certain configuration details, will be for the duration of a transaction, the life of the entity manager, or longer.
  - Any change made to the state of the object while "under management" are **automatically durable**: for example, calling a setter method will result in a SQL update on that entity, perhaps immediately or perhaps just before transaction commit.

Entity State

- **An entity instance that is not actively managed must be in one of three other states, characterized by the object's relationship to a row in the database.**
  - If the object's primary key identifies a row in the database, then the object is **detached** – not managed, but still a representation of that row of data in the database.
  - If there was such a row in the database, but there is no longer, then we're looking at a **removed** entity: the Java object still exists in memory but only as an echo of deleted data.
  - If there will be such a row in the database, then we say the entity instance is in the **new** state, and a future operation will create the corresponding row.
- **Non-managed entities function more or less as normal Java objects, and can be used by code that is not aware of JPA or persistence concerns generally.**

State Transitions

- The following diagram summarizes possible state transitions for entities, as they move through typical CRUD operations:



- We discuss various transitions on the following pages.

State Transitions

- **When you load an entity, either using the find method or by executing a query in whose results the entity is returned, you can get one of two entity states.**
  - If a **transaction** is in force, the entity will be **managed**, so that you can make changes to it and be assured of durability.
  - Otherwise it will be in the **detached** state, which is appropriate for read-only usage.
- **A number of operations will cause a managed entity to become detached:**
  - Calling **detach** on the entity manager for that entity
  - Calling **clear** on the entity manager (affects all managed entities)
  - Calling **close** on the entity manager (affects all managed entities) – excepting the use of the "extended" persistence context in Java EE
  - Completing a **transaction** when the persistence context is transaction-bounded, as it usually is in EE environments
  - **Serializing** and then **de-serializing** the entity itself (existing entity unaffected, new instance produced by de-serialization is detached)

State Transitions

- **A detached entity can be merged into a persistence context, by calling merge on the entity manager.**
  - This ultimately triggers a SQL UPDATE to that database row.
  - This method is unusual in that it does not directly effect a state transition on the given entity.
  - Rather, it **returns a managed entity** based on the detached one you provide; you must be careful to use the returned entity going forward, if you want durability.
- **You can get a new entity into the persistence context in two ways, both of which will trigger a SQL INSERT:**
  - Call **persist** on the entity manager. Any generated IDs will be filled in for you and you can use the existing instance as a managed entity from there.
  - Call **merge** on the entity manager. This method works with detached entities, as above; but it will also work for a new entity. To get the managed entity, again, you must read the return value.
- **Call the remove method to issue a SQL DELETE, and to move the entity into the removed state.**

State Transitions

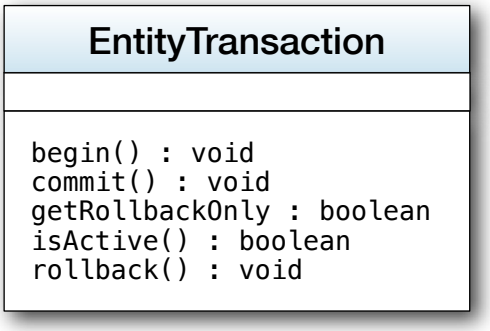
- **A commit during a transaction forces a flush of the persistence context and a commit in the database.**
  - Anytime the persistence provider generates SQL to execute on the database, the persistence context has been flushed.
  - SQL is generated to complete the transaction on the database that corresponds to the entity transaction.
  - Any subsequent operations will incorporate these changes.
- **The refresh method will update all managed entities with state from the database, overwriting potential state changes.**
  - Use care when executing because it is possible to lose uncommitted changes made to an entity.

Transactions

- **JPA applications and components manage transactions either programmatically or declaratively.**
- **In the programmatic approach – also known as an application-managed transaction – we use the EntityTransaction service.**
  - This is not dissimilar to the way it would be done using JDBC or ANSI SQL commands: you call methods to begin, commit, and possibly to roll back your transactions, explicitly.
  - The persistence unit must have the **RESOURCE_LOCAL** transaction type.
  - This approach is more common for Java SE applications, and can be used in EE contexts as well. We'll work with application-managed transactions for this and the next few chapters.
- **In the declarative approach, a container-managed transaction will function to our specifications as provided in metadata.**
  - Here the persistence unit will have the **JTA** ttransaction type, and the entity manager will not allow use of **EntityTransaction**.
  - Instead, the component will use the JTA **@Transactional** or possibly other annotations to set policies for starting, propagating, isolating, and committing transactions.

Programmatic Transaction Control

- Call **getEntityTransaction** on the entity manager for access to the **EntityTransaction**.



- **begin** and **commit** start and end a transaction, respectively.
  - A **RollbackException** will be thrown if **commit** fails.
  - To avoid a **RollbackException**, call the **getRollbackOnly** method to determine if the transaction is in a failed state.
- If necessary, the transaction can be rolled back using **rollback**.
  - A **PersistenceException** will be thrown if **rollback** fails.
- **isActive** returns true if a transaction is active.
  - If you attempt to start a new transaction while **isActive** is true, an **IllegalStateException** will be thrown.
  - We will see a use of the **isActive** method on the next page.

Programmatic Transaction Control

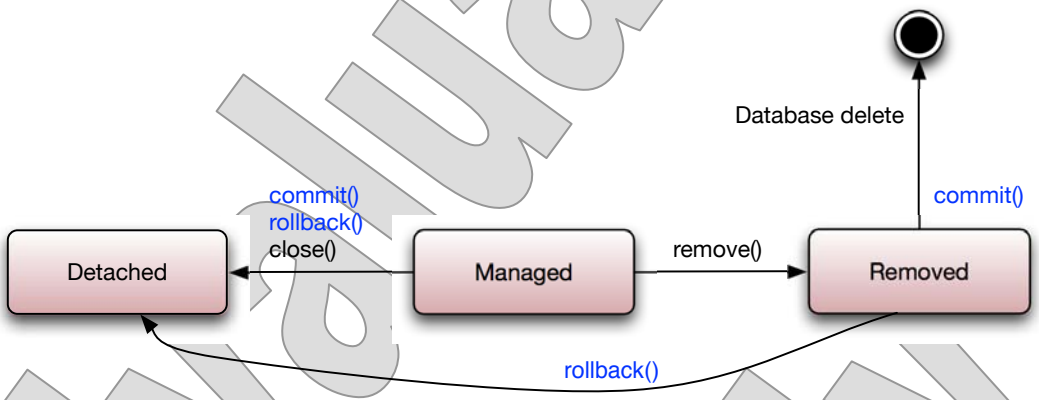
- This leads to a typical code structure involving a system of **try/catch/finally** blocks, to control the entity manager's and transaction's lifecycles:

```
EntityManager em = null;
EntityTransaction et = null;
try
{
  em = emf.createEntityManager ();
  et = em.getEntityTransaction ();
  et.begin ();
  // Carry out operations here
  et.commit ();
}
catch (Exception ex) // or more specific type(s)
{
  if (et != null && et.isActive ())
    et.rollback ();
  // Logging and other handling here
}
finally
{
  if (em != null)
    em.close ();
}
```

  - There are a few variants on this theme, such as working with a provided entity manager (and most likely not closing it), or catching optimistic-locking exceptions.
  - We'll see plenty more examples as we move ahead.
  - Sadly, as of JPA 2.1, neither **EntityManagerFactory** nor **EntityManager** implements the Java-7 **AutoCloseable** interface, and so try-with-resources, so helpful in JDBC, is not an option.

Transactions and Entity States

- CRUD operations require as much care as they would if you were using SQL directly, or JDBC.
  - Database **constraints**, especially foreign-key and not-null constraints, will trigger an exception if they are violated.
  - Updating or deleting entities that have **relationships** can be especially troublesome, until you gain some experience.
- You must use **EntityManager** operations in the context of a transaction as outlined on the next page if they mutate data.



- Transactional operations must be used with **managed** entities to have entity operations coordinated with database operations.
- Either a **commit** or **rollback** call can result in managed entities becoming **detached** as a transaction-bounded persistence context will then be closed.
- If an entity has been removed, pending a database delete, commit will result in its deletion while rollback will result instead in detaching the entity.
  - This accords with the definition of a detached entity as one that still relates by primary key to a row in the database.

Earthlings in Action

DEMO

- **CRUD operations involve managed entities that typically have relationships with other entities.**
  - The operations have to respect underlying constraints and relationships or they will throw an exception.
  - All entities and their associated entities should be managed.
- **We'll work in Earthlings_Step5 and try a few entity operations.**
  - The completed demo is in **Earthlings_Step6**.
- **src/cc/hr/client/EmployeeManager.java shows us what might be a typical business operation: add a new employee and new job to an existing department.**
- **We call the method in the main method, after booting up an entity manager factory and an entity manager:**

```
emf = createEntityManagerFactory("EarthlingsPU");
em = emf.createEntityManager();
...
addEmployeeAndJob(em, "Provost", "Marshal",
  Gender.M, "Security", 80000, 110000);
```

- **As shown on the following page, this method ...**
  - Creates and populates a **Job**
  - Creates and populates an **Employee**, including the embedded **Address**
  - Calls **em.persist** to add both objects to the database

Earthlings in Action

DEMO

```
public static void addEmployeeAndJob
  (EntityManager em, String firstName,
   String lastName, Gender gender,
   String jobName, long minimumSalary,
   long maximumSalary)
{
  em.clear();
  EntityTransaction et = em.getTransaction();

  et.begin();
  Job job = new Job
    (jobName, minimumSalary, maximumSalary);
  Employee emp = new Employee
    (firstName, lastName, gender,
     em.find(Department.class, 1), job);
  Address adr = new Address();
  adr.setState(State.GA);
  emp.setAddress(adr);

  em.persist(emp);
  em.persist(job);
  et.commit();

  et.begin();
  em.remove(emp);
  em.remove(job);
  et.commit();
  ...
}
```

Earthlings in Action

DEMO

1. Run **cc.hr.client.EmployeeManager** as a Java application.

```
Adding Provost Marshal and Security
Removed Provost Marshal, Security
```

   – No problem.

2. But, edit **src/META-INF/persistence.xml** to use Hibernate, by copying the prepared <provider> element from the code comment and pasting outside the comment:

```
<provider>org.hibernate.ejb.HibernatePersistence
  </provider>
```

3. Test again, and ...

```
Adding Provost Marshal and Security
WARN: HHH000437: Attempting to save one or more
entities that have a non-nullable association with
an unsaved transient entity. The unsaved transient
entity must be saved in an operation prior to
saving these dependent entities.
```

   – In other words, when we attempted to persist **emp**, the **primary key** (ID property) of the **job** object was **null**.

   – Hibernate tried to use this value as the **foreign key**, which is required in the EMPLOYEES table, and failed at that point.

   – Since persist triggers the a SQL INSERT, persisting **job** first is important because the database will only create the JOBS primary key after the row is inserted in the database.

   – Then that ID will be filled in on the **job** object by the provider, and be available as the foreign-key value for the second INSERT.

Earthlings in Action

DEMO

- **Both entity transactions and database transactions are units of work, and both have to complete successfully or rollback.**
- **As bad as we may feel that we still have to consider the underlying database, it is often unavoidable when it comes time to debug a problem.**
  - So, why did this work cleanly under EclipseLink?
  - A JPA provider may **change the sequence** of operations against the underlying database. This can allow it to handle some out-of-order JPA operations gracefully.
  - It can also be a bit disconcerting, as if it's re-writing your code. But, love it or hate it, this is a compliant behavior under JPA.
  - In any case, code such as the current method should not be considered safe, and we can see it is not portable to Hibernate, if nothing else.

Earthlings in Action

DEMO

4. Fix the method by exchanging the two calls to **persist**:

```
et.begin();
Job job = new Job(jobName, minimumSalary,
  maximumSalary);
Employee emp = new Employee(firstName, lastName,
  gender, em.find(Department.class, deptId),
  job);
em.persist(job);
em.persist(emp);
et.commit();
```

- **The code above now works correctly, with any JPA provider.**

```
Adding Provost Marshal and Security
Removed Provost Marshal, Security
```

  - Notice too that the remove operations that are left in here (to clean up and make this application's behaviors repeatable for testing) are already in the correct order: first in, last out ...

```
et.begin();
em.remove(emp);
em.remove(job);
et.commit();
```

  - There are a few other test operations of a similar nature on this class, whose code may be worth a quick look.
  - Of these note that **addEmployeeAndJobToDepartment** has the same bug in it that we just fixed, with two **persist** calls out of order.

Creating Queries

- **In addition to the find method, the entity manager gives you a handful of ways in which to create Query objects, which then can be executed and their results processed.**
- Create **dynamic queries** using the **Java Persistence Query Language**, or **JPQL**.
  - This is a **SQL-like** language that is also **object-oriented**, speaking in terms of entities and properties, rather than tables and columns.
  - Call **createQuery** to parse a JPQL string into a structured query.

```
List<Job> results = em.createQuery
  ("select j from Job j", Job.class);
```

  - You can immediately execute and process results, with **getResultList** for possibly multiple results, or with **getSingleResult** when you're certain to get exactly one row.
  - The **TypedQuery** object that you get from **createQuery** also supports other methods, in a fluent-API design that allows you to chain calls that fine-tune the query before executing it.
  - For example you can call **setParameter** to fix the value of a parameter in the JPQL string; or call **setFirstResult** and **setMaxResults** to execute a range query.
  - A concern with using dynamic queries is the potential for **JPQL injection** attacks when the query is triggered via a user interface.
- **JPA also has a Java-based query-building API known as the Criteria API, which we will discuss in a later chapter.**

Normalizing Payroll

LAB 4

Suggested time: 30 minutes

In this lab you will create a console application that runs a simple update algorithm over the Earthlings database. You will execute a query for all employees whose salaries are outside of the range prescribed by their job descriptions, and adjust to put them in tolerance.

Detailed instructions are found at the end of the chapter.

Named Queries

- **When you see potential for a query to be re-used, you can encapsulate it as a named query.**
- **Place a @NamedQuery annotation on any entity in the persistence unit.**

```
@NamedQuery(name="Job.getAllJobs",
  query="select j from Job j");
@Table(name="JOBS")
public class Job
```

  - Often you'll annotate the entity that is the query's **result type**.
  - But named queries are **global** to the persistence unit, so actually any entity will do – and sometimes there is no one, obvious choice.
  - The name must be unique within the whole persistence unit, too, and one convention is to include the **entity class name** and then a **dot** to prefix the query name.
  - After the prefix, a **descriptive name** for the query appears.

- **To execute a named query, use the createNamedQuery method on the entity manager.**

```
List<Job> jobs = em.createNamedQuery
  ("Job.getAllJobs", Job.class).getResultList();
```

  - Notice how similar is this usage to our previous calls to **createQuery** – and, indeed, the compiler won't catch it if you use one of these methods in place of the other.
  - At runtime, the first argument to **createNamedQuery** is a name, to be found amongst the named queries for the persistence unit – whereas on **createQuery** it's the JPQL itself.

Query Parameters

- **A query can include replaceable parameters, of two types:**
  - **Named parameters**, stated as a colon followed by a name
  - **Positional parameters**, stated as a number followed by a question mark – for example **1?**
- **Any query can do this, but parameters are especially common on named queries, since they are meant for re-use.**

```
@NamedQuery(name = "Job.getAllJobsById",
   query = "select j from Job j where j.id = :id");
```

  - In this query, we are able to select a single **Job** by its primary key.
- **Supply parameters to such a query after creating it and before executing it:**

```
Job job =
  em.createNamedQuery("Job.getJobById", Job.class)
    .setParameter("id", id);
    .getSingleResult();
```

  - In the method above, we pass the ID to the query using the **setParameter** method.
  - This kind of query is safer than a dynamic query because it is not subject to a JPQL injection attack; its structure is **static** and the parameters cannot alter that structure, only populate it.
- **Notice that since this query can only return a single value, we have used the getSingleResult method.**

Native Queries

- **In addition to the query and named query, JPQL also supports a native query, or SQL query, when needed.**
  - This provides a final fallback for situations where JPQL is insufficient, in its overall expressiveness or when trying to use proprietary database extensions.
  - Native queries are very similar to their JPQL counterparts.

```
private static final String query =
  "select * from licenses";
List<?> jobs = entityManager
  .createNativeQuery(query, License.class)
  .getResultList();
```

  - This method always returns the weakly-typed **Query**; it has had this method signature since JPA 1.0 and so there was not the opportunity to define a strongly-typed overload as we've seen with **createQuery**.
- **Before going too far with native queries, note that there are often more attractive alternatives.**
  - Some JPA providers have extended features geared to one or more database systems, so you might be able to work with a more specific API instead of with native SQL.
  - For example, EclipseLink programmers may want to investigate its numerous Oracle database extensions.
  - Also, we'll see in a moment some JPA-2.1 features that make it easier to take advantage of SQL extensions without leaving the more portable, object-oriented JPA space.

Named, Native Queries

- We may as well complete the square: yes, you can define a **@NamedNativeQuery**, and instantiate it with **createNamedNativeQuery** on the entity manager.

```
@NamedNativeQuery(name="Drug.getDrugBySoundex",
  query="select * from pharmacy.drugs "
      + "where soundex(drug_name) = soundex(?)",
  resultClass = Drug.class);
...
List<Drug> drugs = em.createNamedQuery
  ("Drug.getDrugBySoundex", Drug.class)
    .setParameter(1, "welbutren").getResultList();
```

  - This method will return a **TypedQuery** based on the class that you pass in as the second argument, as **createQuery** will do.

Named Queries

EXAMPLE

- In **Earthlings_Step7**, a new client application exercises a named query that can read the total payroll of a given department.
- First, see **src/cc/hr/entity/Department.java**.

```
@NamedQueries
({
 @NamedQuery(name="Department.getPayrollByName",
  query="select sum(e.salary) from Department d " +
   "inner join d.employees e where d.name= :name"),
 @NamedQuery(name=
   "Department.getDepartmentNamesAndPayroll",
  query="select d.name, sum(e.salary) from " +
   "Department d inner join d.employees e " +
   "group by d.name order by d.name"),
 @NamedQuery(name=
   "Department.getDepartmentsByNameAndPayroll",
  query="select d, sum(e.salary) from " +
    "Department d inner join d.employees e " +
    "group by d order by d.name")
})
public class Department
```

  - We use the common trick with Java annotations of grouping them in a **plural** version of the primary annotation whose value is an array of the primary, **singular** annotation: in this case, a single **@NamedQueries** that collects **@NamedQuery** definitions.
  - Each of the three queries has a distinct name, but all qualified by **Department** as the entity type most intuitively associated with what the queries do.
  - Note that the first query, "Department.getPayrollByName", takes a single, **named parameter**.

Named Queries

EXAMPLE

- In **src/cc/hr/client/TestNamedQueries.java**, a helper method **getPayrollByDepartment** invokes that first query, propagating its own parameter as the query parameter:

```
static Long getPayrollByDepartment
  (EntityManager em, String name)
{
  try
  {
    return (Long) em.createNamedQuery
      ("Department.getPayrollByName")
    .setParameter("name", name)
    .getSingleResult();
  }
```

  - The **main** method then calls this helper method three times, with a different department name each time:

```
emf = Persistence.createEntityManagerFactory
  ("EarthlingsPU");
em = emf.createEntityManager();
ProviderUtil.reportProvider(em);

System.out.println ("Department payrolls:");
System.out.format("%-20s %,10d%n",
  "Research",
  getPayrollByDepartment
    (em, "Research"));
System.out.format("%-20s %,10d%n",
  "Software Development",
  getPayrollByDepartment
    (em, "Software Development"));
System.out.format("%-20s %,10d%n",
  "Test And Integration",
  getPayrollByDepartment
    (em, "Test And Integration"));
```

Named Queries

EXAMPLE

- **Run this class as a Java application, and see the results for each of the three dpeartments:**

```
Department payrolls:
Research                1,035,000
Software Development      906,000
Test And Integration      533,000
```

Stored-Procedure Queries

JPA 2.1

- JPA 2.1 introduces a standard means of invoking stored procedures.
  - This was possible only via certain providers' proprietary APIs under JPA 2.0 and earlier.
- Call **createStoredProcedureQuery** on the entity manager, providing the name of the stored procedure in as defined in the target schema.

```
StoredProcedureQuery query =
  em.createStoredProcedureQuery ("SP");
query.setParameter (1, 4);
query.registerStoredProcedureParameter
  (2, String.class, ParameterMode.OUT);
query.execute ();
return query.getOutputParameterValue(2);
```

  - You can **set parameters**, as you would for a JPQL query.
  - Beware that not all JDBC drivers support **named parameters** for stored procedures, so you may need to rely on **positional parameters** instead, as shown above.
  - Because a stored procedure doesn't have "results" or a return type, you don't get an entity or list of entities back when you **execute**.
  - Instead, call **registerStoredProcedureParameter** for "out parameters", and then you can read the provided value with **getOutputParameterValue**.

Named, Stored-Procedure Queries

JPA 2.1

- Because there can be so much involved in even a minimal invocation of a stored procedure, the option to encapsulate the query as a named query is especially appealing.
- Use **@NamedStoredProcedureQuery**:

```
@NamedStoredProcedureQuery
 (name="MyEntity.SP", procedureName="SP",
  parameters=
  { @StoredProcedureParameter (name="numberIn",
     type=Integer.class, mode=ParameterMode.IN),
    @StoredProcedureParameter (name="stringOut",
     type=String.class, mode=ParameterMode.OUT) })
public class Employee
```

  - Note that in this usage we're not pinning our hopes on support for the the parameter names shown above; the array of **@StoredProcedureParameter** annotations implies ordinal position for each parameter, and that will be used.

- Then create and use the query like this:

```
StoredProcedureQuery query =
  em.createNamedStoredProcedureQuery
    ("MyEntity.SP");
query.setParameter (1, 4);
query.execute ();
return query.getOutputParameterValue(2);
```

  - Notice that, while you still have to provide values for input parameters, the registration of output parameters is entirely encapsulated in the named-query annotation, so that it doesn't need to be done over and over in various parts of the application.

Named, Native, Stored-Procedure Queries

- Just kidding.

Normalizing Payroll, Again

DEMO

- **This chapter's lab was, hopefully, a useful exercise in working with entity managers and carrying out basic persistence operations.**
- **But, as it happens, the Earthlings schema already defines a stored procedure called normalizesalaries, and we'd do better to take advantage of that.**
  - Do your work in **Earthlings_Step8**, where the old code has already been ripped out in favor of a stored-procedure call.
  - The completed demo in **Earthlings_Step9** goes a step farther by capturing this bit of logic as a named, stored-procedure query.
- **The stored procedure is implemented, quite differently, for Derby and Oracle versions of the schema.**
  - It's already in place, having been set up as part of creating the schema along with all the usual DDL and SQL INSERTs.
  - We won't dive into the particulars of how the procedure is defined, but if you're curious you can see the Derby source code under **Schema/Earthlings/Derby**, in **src/cc/util/earthlings.DerbyProcedures**.
  - For Oracle, the PL/SQL definition is found in **Schema/Earthlings/Oracle/earthlings_appcreate_oracle.sql**.

Normalizing Payroll, Again

DEMO

1. Open **src/cc/hr/client/NormalizePayroll.java**, and see that the **main** method has much less code in it than when you left it at the end of the lab exercise:

```
em.getTransaction ().begin ();
StoredProcedureQuery spq =
  em.createStoredProcedureQuery
    ("normalizeSalaries");
spq.registerStoredProcedureParameter
    (1, String.class, ParameterMode.OUT);
spq.execute ();
System.out.println
  (spq.getOutputParameterValue (1));
em.getTransaction ().commit ();
```

   – The stored procedure fills the output parameter with a formatted string that is more or less equivalent to the program output from the previous version.

2. Run this class as a Java application, and see that it carries out the same algorithm, such that it adjusts the salaries of five employees:

```
Normalizing all salaries ...

Salary for Ross Franks was 68000 –
  adjusted to 70000
Salary for John Bigboote was 95000 –
  adjusted to 90000
Salary for Penny Pretty was 24000 –
  adjusted to 25000
Salary for Devin Smythe was 64000 –
  adjusted to 60000
Salary for Audra Swanson was 72000 –
  adjusted to 65000

Done.
```

Normalizing Payroll, Again

DEMO

3. Open **src/cc/hr/entity/Employee.java** and add the following annotation to the class, to define the named, stored-procedure query:

```
@NamedStoredProcedureQuery
  (name = "Employee.normalizeSalaries",
   procedureName="normalizeSalaries",
   parameters=@StoredProcedureParameter
    (type=String.class, mode=ParameterMode.OUT))
public class Employee
```

4. In **NormalizePayroll.java**, change over to use this named query:

```
StoredProcedureQuery spq =
  em.createNamedStoredProcedureQuery
    ("Employee.normalizeSalaries");
spq.registerStoredProcedureParameter
  (1, String.class, ParameterMode.OUT);
spq.execute ();
System.out.println
  (spq.getOutputParameterValue (1));
```

   – We no longer need to call **registerStoredProcedureParameter**, as this is now part of the named-query definition.

5. In order to see results, you will need to run **recreatedb** from the **Schema/Earthlings/**(**Derby** or **Oracle**) directory.

6. Then, run the application again, and see that it performs in the same way.

SUMMARY

- **In this chapter, we looked at CRUD operations on our completed entities.**
  - We haven't exhausted the possibilities of enhancing our entities with additional annotations.
  - Later we will discuss **validation** and **cascade** annotations.
- **We started out examining the EntityManager interface and the operations we can perform on that interface.**
- **We moved to EntityTransaction next, and found that we cannot forget our underlying database when performing a unit of work.**
- **Entity state is very important to understand because transactional operations are limited to managed entities.**
  - There are a number of conditions that will cause an entity to become detached, or unmanaged.
  - We will examine entity state in more detail in a later chapter.
- **CRUD operations take a lot of care, to get them to work the way you expect.**
  - This is an area where prior SQL programming experience can be helpful.
- **We will go on to learn JPQL in depth in the next chapter.**
  - This will allow us to perform sophisticated queries on our entities, either for CRUD operations, reporting, or interactive web applications.

Normalizing Payroll

LAB 4

In this lab you will create a console application that runs a simple update algorithm over the Earthlings database. You will execute a query for all employees whose salaries are outside of the range prescribed by their job descriptions, and adjust to put them in tolerance.

Lab project: **Earthlings_Step6**

Answer project: **Earthlings_Step7**

Files: * to be created
src/cc/hr/client/NormalizePayroll.java *

Instructions:

1. Create a new class **cc.hr.client.NormalizePayroll**. You can use the **main** method from the demo's **EmployeeManager.java** as a template for this one. keep the initialization of an entity manager factory and an entity manager, and keep the **try/catch/finally** system and the error-handling and cleanup code. Just clean out the **try** block after the **em** variable is assigned.
2. Call **em.createQuery**, passing the following query string and **Employee.class** as arguments. This will give you a **TypedQuery<Employee>**, which you can store in a local variable **query**.

```
select e from Employee e
where e.salary not between e.job.minimumSalary and e.job.maximumSalary
```

3. Call **query.getResultList** and enter a loop over all results, which will be of type **Employee**.
4. For each result, get the **job** property as a local variable **job**.
5. Declare a local variable **newSalary**, of type **long**.
6. If the employee's **salary** is less than the job's **minimumSalary**, set **newSalary** to the **minimumSalary** value.
7. Otherwise, if the **salary** is greater than the **maximumSalary**, set **newSalary** to the **maximumSalary** value.
8. Print a line to the console showing the employee's name, old salary, and adjusted salary.

Normalizing Payroll LAB 4

9. Set the employee salary to the **newSalary**.
10. Test your application. For this lab, since you will be doing updates, you should do three things in sequence: run **recreatedb** from the appropriate subdirectory of **Schema/Earthlings** (that is, either for Derby or Oracle); then **run NormalizePayroll**, and you should see output something like this:

```
Normalizing all salaries ...

Salary for Ross Franks was $68,000 -- adjusted to $70,000.
Salary for John Bigboote was $95,000 -- adjusted to $90,000.
Salary for Penny Pretty was $24,000 -- adjusted to $25,000.
Salary for Devin Smythe was $64,000 -- adjusted to $60,000.
Salary for Audra Swanson was $72,000 -- adjusted to $65,000.

Done.
```

Then run it again. If your updates worked, then you should see no changes made the second time – something like this:

```
Normalizing all salaries ...

Done.
```

Ah ... but you don't, do you? It keeps showing the same output every time you run the application. Are your changes actually durable? Do you know what's missing?

Remember that entities are in the managed state when read from the database only if a transaction is in force. In this case, no transaction is active, and so the query is legal, and the changes to the entities are legal – but they are not durable, because the entities themselves are in the Detached state. This is one odd side effect of transparent entity management: you can get these false-positive results from your code, and come away thinking you've made a change to the database when you have not.

11. Before creating your query, call **em.getTransaction** and then call **begin** on the **EntityTransaction** object that you get back.
12. After your loop, call **commit** on that same object.
13. In the **catch** block, after calling the exception-handling helper method, check to see if the transaction **isActive**. If so, call **rollback** on it.
14. Test again. Now you should see clean results when you re-initialize the database and run hour application twice: the lack of anything to do on the second run proves that the changes made in the first run were durable.