MANAGING ENTITIES

Java Persistence API has two sides. The first is the ability to map objects to a relational database. Configuration by exception allows persistence providers to do most of the work without much code, but the richness of JPA also allows customised mapping from objects to tables using either annotation or XML descriptors. From simple mapping (changing the name of a column) to more complex mapping (inheritance), JPA offers a wide spectrum of customisations. As a result, you can map almost any object model to a legacy database.

The other aspect of JPA is the ability to query these mapped objects. In JPA, the centralised service for manipulating instances of entities is the entity manager. It provides an API to create, find, remove, and synchronise objects with the database. It also allows the execution of different sorts of JPQL queries, such as dynamic, static, or native queries, against entities. Locking mechanisms are also possible with the entity manager.

The code in this chapter can be found at https://github.com/agoncal/agoncal-fascicle-jpa/tree/2.2/managing

Entity Management APIs

Before we look at how to manage entities, let's first have a brief overview of APIs used to manage entities. Figure 1 shows the main interfaces which are located in three different packages. Under the root package `javax.persistence` we find everything related to obtaining an `EntityManager`, transactions and queries. The sub-package `javax.persistence.criteria` hosts the interfaces related to criteria queries, while the sub-package `javax.persistence.metamodel` is related to the entity metamodel.

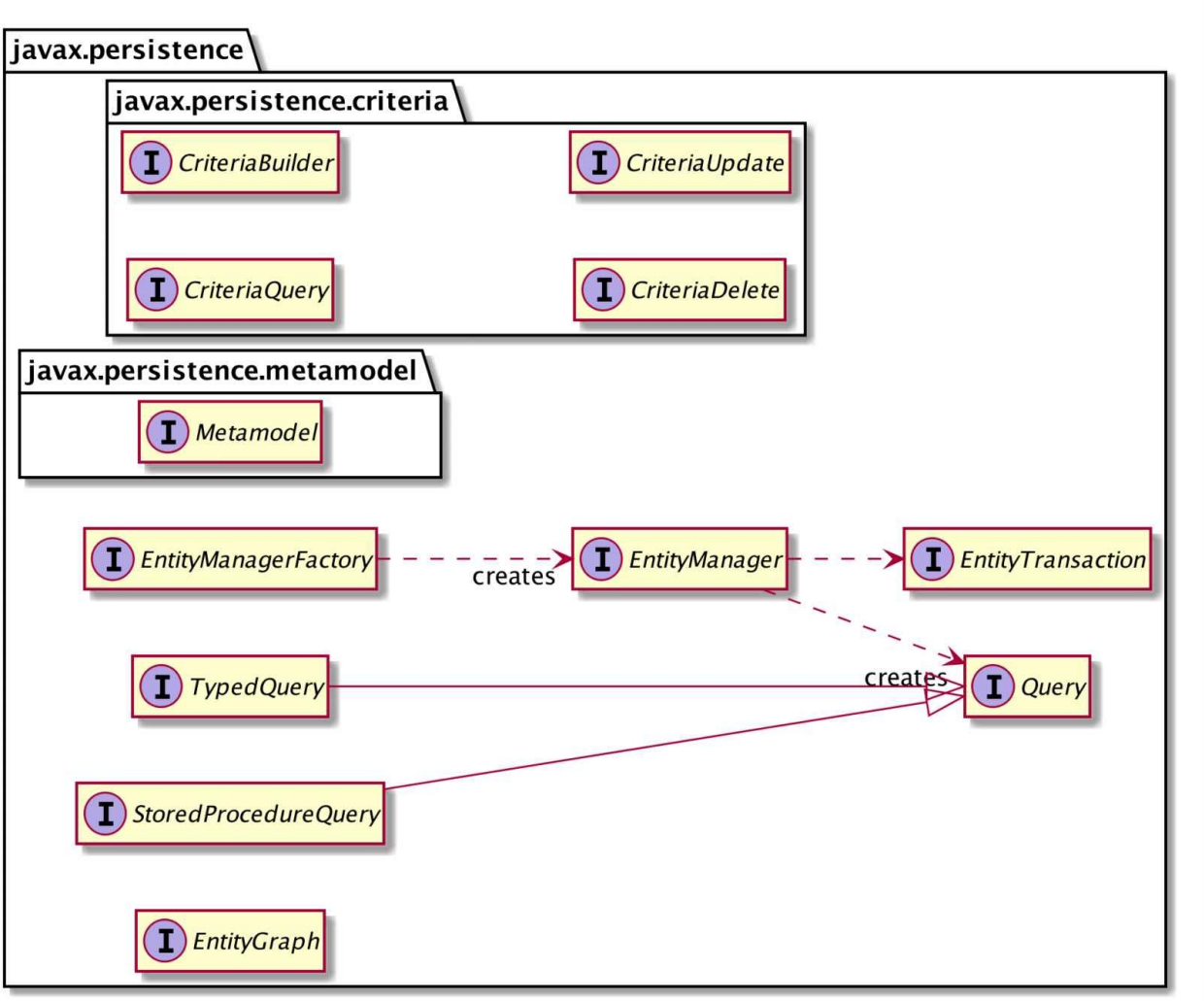

Figure 1. Entity management APIs

Let's dig into these APIs.

Entity Manager API

The entity manager manages the state and life cycle of entities as well as querying entities within a persistence context. The entity manager is responsible for creating and removing persistent entity instances and finding entities by their primary key. It can lock entities in order to protect against concurrent access by using optimistic or pessimistic locking and can use JPQL queries to

retrieve entities following certain criteria (more on concurrency and locks in Chapter 8).

When an entity manager obtains a reference to an entity, the entity is said to be "managed." Up to that point, the entity is seen as a regular POJO (i.e. detached). The strength of JPA is that entities can be used as regular objects by different layers of an application and become managed by the entity manager when you need to load or insert data into the database. When an entity is managed, you can carry out persistence operations, and the entity manager will automatically synchronise the state of the entity with the database. When the entity is detached (i.e. not managed), it returns to a simple POJO and can then be used by other layers (e.g. the presentation layer) without synchronising its state with the database.

With regard to persistence, the real work begins with the entity manager. `EntityManager` is an interface implemented by a persistence provider that will generate and execute SQL statements. The `javax.persistence.EntityManager` interface provides the API to manipulate entities (subset shown in Listing 1).

Listing 1. Subset of the EntityManager API

```
public interface EntityManager {

  // Factory to create an entity manager, close it and check if it's
open
  public void close();
  public boolean isOpen();
  public EntityManagerFactory getEntityManagerFactory();

  // Return the underlying provider object for the EntityManager
```

```
  public <T> T unwrap(Class<T> cls);
  public Object getDelegate();

  // Sets and gets an entity manager property or hint
  public void setProperty(String propertyName, Object value);
  public Map<String, Object> getProperties();
```

The entity manager can be seen as a first-level cache for entities. Therefore, it has a few methods to check if an entity is in the first-level cache (a.k.a. managed), flush the cache to the database or refresh the cache from the database. The methods `clear` and `detach` remove an entity from the entity manager.

```
// Synchronises the persistence context to the underlying database
public void flush();
public void setFlushMode(FlushModeType flushMode);
public FlushModeType getFlushMode();

// Refreshes the state of the entity from the database, overwriting
any changes made
public void refresh(Object entity);
public void refresh(Object entity, LockModeType lockMode);

// Clears the persistence context and checks if it contains an
entity
public void clear();
public void detach(Object entity);
public boolean contains(Object entity);
```

For basic CRUD operations, the entity manager has a few methods that allow you to persist, merge, remove and find entities by their primary key.

```
// Persists, merges, removes and finds an entity to/from the
database
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 <T> T getReference(Class<T> entityClass, Object primaryKey);
```

When CRUD is not enough, JPA allows you to create a certain number of queries using JPQL, native SQL, criteria queries or stored procedures.

```
  // Creates an instance of Query or TypedQuery for executing a JPQL
statement
  public Query createQuery(String qlString);
  public <T> TypedQuery<T> createQuery(String qlString, Class<T>
resultClass);

  // Creates an instance of Query or TypedQuery for executing a
named query
  public Query createNamedQuery(String name);
  public <T> TypedQuery<T> createNamedQuery(String name, Class<T>
resultClass);

  // Creates an instance of Query for executing a native SQL query
  public Query createNativeQuery(String sqlString);
  public Query createNativeQuery(String sqlString, Class
resultClass);
  public Query createNativeQuery(String sqlString, String
resultSetMapping);

  // Creates a StoredProcedureQuery for executing a stored procedure
in the database
  public StoredProcedureQuery createNamedStoredProcedureQuery(String
name);
  public StoredProcedureQuery createStoredProcedureQuery(String
procedureName);
  public StoredProcedureQuery createStoredProcedureQuery(
    String procedureName, Class... resultClasses);
  public StoredProcedureQuery createStoredProcedureQuery(
    String procedureName, String... resultSetMappings);

  // Metamodel and criteria builder for criteria queries (select,
update and delete)
```

```
  public CriteriaBuilder getCriteriaBuilder();
  public Metamodel getMetamodel();
  public <T> TypedQuery<T> createQuery(CriteriaQuery<T>
criteriaQuery);
  public Query createQuery(CriteriaUpdate updateQuery);
  public Query createQuery(CriteriaDelete deleteQuery);

  // Creates and returns an entity graph
  public <T> EntityGraph<T> createEntityGraph(Class<T> rootType);
  public EntityGraph<?> createEntityGraph(String graphName);
  public  EntityGraph<?> getEntityGraph(String graphName);
  public <T> List<EntityGraph<? super T>> getEntityGraphs(Class<T>
entityClass);
}
```

The entity manager has a few methods that let you obtain a transaction or join an existing one. It also lets you lock an entity with either optimistic or pessimistic locks.

```
// Returns an entity transaction
public EntityTransaction getTransaction();

// Indicates if a JTA transaction is active and joins the
persistence context to it
public void joinTransaction();
public boolean isJoinedToTransaction();

// Locks an entity with the specified lock mode type (optimistic,
pessimistic...)
public void lock(Object entity, LockModeType lockMode);
public LockModeType getLockMode(Object entity);
```

Don’t get scared by the API in Listing 1, as this chapter covers most of the methods.

Obtaining an Entity Manager

The entity manager is the central interface used to interact with entities, but it first has to be obtained by the application. Depending on whether it is a container-managed environment (e.g. see Spring or CDI on Chapter 9) or an application-managed environment, the code can be quite different. For example, in a container-managed environment, the transactions are managed by the container. That means you don't need to explicitly write the commit or rollback, which you have to do in an application-managed environment.

The term *application managed* means the application is responsible for explicitly obtaining an instance of `EntityManager` and managing its life cycle (creating and then closing the entity manager programmatically). The code in Listing 2 demonstrates how a class running in a Java SE environment gets an instance of an entity manager. It uses the `Persistence` class to bootstrap an `EntityManagerFactory` associated with a persistence unit (`cdbookstorePU`), which is then used to create an entity manager. Notice that, in an application-managed environment, the developer is responsible for creating and closing the entity manager (i.e. managing its life cycle). It will throw a `javax.persistence.PersistenceException` if it cannot create or close the entity manager.

Listing 2. Obtaining an EntityManager within an Application-managed Component

```
public class BookService {

  public void createBook() {

    // Obtains an entity manager
    EntityManagerFactory emf =
```

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

    // Obtains a transaction
    EntityTransaction tx = em.getTransaction();

    // Creates an instance of book
    Book book = new Book().title("H2G2").price(12.5F).isbn("1-84023-
742-2").nbOfPages(354);

    // Persists the book to the database
    tx.begin();
    em.persist(book);
    tx.commit();

    // Closes the entity manager and the factory
    em.close();
    emf.close();
  }
}
```

In a container-managed environment, the way to acquire an entity manager is through injection. Depending on the container, injection can happen with the `@PersistenceContext` annotation or `@Inject` (more on injection in Chapter 9). Also, we don't need to programmatically create or close the entity manager, as its life cycle is managed by the container. Listing 3 shows the code of a transactional service (see the `@Transactional` annotation) into which the container injects a reference of the `cdbookstorePU` persistence unit.

Listing 3. Obtaining an EntityManager within a Container-managed Component

```
@Transactional
public class BookService {

  // Obtains an entity manager
```

```
  @PersistenceContext(unitName = "cdbookstorePU")
  private EntityManager em;

  public void createBook() {

    // Creates an instance of book
    Book book = new Book().title("H2G2").price(12.5F).isbn("1-84023-
742-2").nbOfPages(354);

    // Persists the book to the database
    em.persist(book);
  }
}
```

Compared with Listing 2, the code in Listing 3 is much simpler. First, there is no `Persistence` or `EntityManagerFactory` as the container injects the entity manager instance. The application is not responsible for managing the life cycle of the `EntityManager` (creating and closing the `EntityManager`). Second, because transactional beans manage the transactions, there is no explicit commit or rollback. Chapter 9 shows how to integrate Java Persistence API with container-managed technologies such as CDI and Spring.

Persistence Context

Before exploring the `EntityManager` API in detail, you need to understand a crucial concept: the *persistence context*. A persistence context is a set of managed entity instances at a given time for a given user's transaction: only one entity instance with the same persistent identity can exist in a persistence context. For example, if a `Book` instance with an Id of 12 exists in the persistence context, no other book with this ID can exist within that same persistence

context. Only entities that are contained in the persistence context are managed by the entity manager, meaning that changes will be reflected in the database.

The entity manager updates or consults the persistence context whenever a method of the `javax.persistence.EntityManager` interface is called. For example, when a `persist()` method is called, the entity passed as an argument will be added to the persistence context if it doesn't already exist. Similarly, when an entity is found by its primary key, the entity manager first checks whether the requested entity is already present in the persistence context. The persistence context can be seen as a first-level cache. It's a short-lived space where the entity manager stores entities before flushing the content to the database. By default, objects just live in the persistent context for the duration of the transaction.

To summarise, let's look at Figure 2 where two users need to access entities whose data is stored in the database. Each user has his their persistence context that lasts for the duration of their own transaction. User 1 gets the `Book` entities with IDs equal to 12 and 56 from the database, so both books get stored in their persistence context. User 2 gets the entities 12 and 34. As you can see, both entities with ID = 12 are stored in each user's persistence context. While the transaction runs, the persistence context acts like a first-level cache, storing the entities that can be managed by the `EntityManager`. Once the transaction ends, the persistence context ends and the entities are cleared.

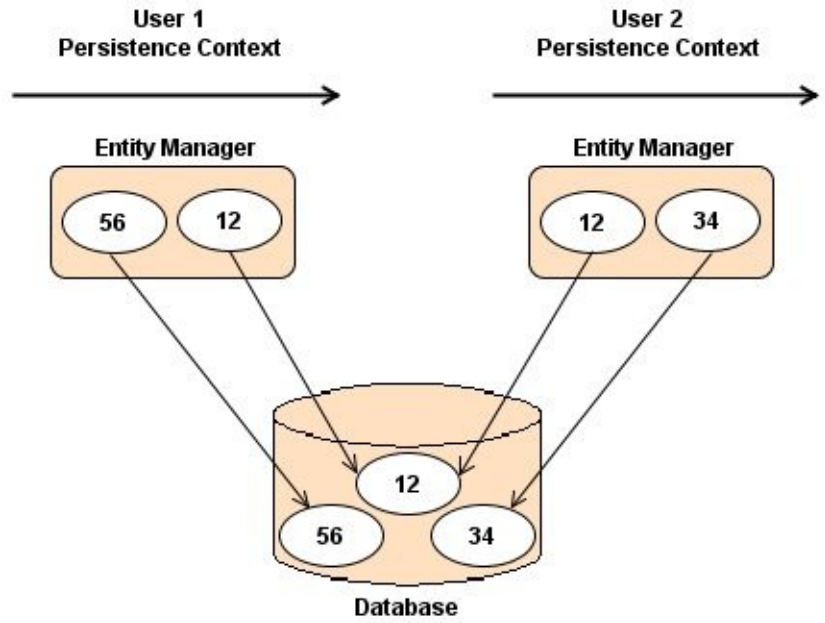

Figure 2. Entities living in different users' persistence context

The configuration for an entity manager is bound to the factory that created it. Whether application or container managed, the factory needs a persistence unit from which to create an entity manager. A persistence unit dictates the settings to connect to the database and the list of entities that can be managed in a persistence context. The `persistence.xml` file (see Listing 4) located in the `META-INF` directory defines a persistence unit. The persistence unit has a name (`cdbookstorePU`) and a set of attributes.

Listing 4. A Persistence Unit with a Set of Manageable Entities

```
<persistence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xmlns="http://xmlns.jcp.org/xml/ns/persistence"

xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd"
             version="2.2">

  <persistence-unit name="cdbookstorePU" transaction-
type="RESOURCE_LOCAL">
```

```
<provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>org.agoncal.fascicle.jpa.managing.Book</class>
    <properties>
      <property name="javax.persistence.schema-
generation.database.action" value="drop-and-create"/>
      <property name="javax.persistence.schema-
generation.scripts.action" value="drop-and-create"/>
      <property name="javax.persistence.schema-
generation.scripts.create-target" value="cdbookstoreCreate.ddl"/>
      <property name="javax.persistence.schema-
generation.scripts.drop-target" value="cdbookstoreDrop.ddl"/>
      <property name="javax.persistence.jdbc.driver"
value="org.h2.Driver"/>
      <property name="javax.persistence.jdbc.url"
value="jdbc:h2:mem:cdbookstoreDB"/>
    </properties>
  </persistence-unit>
</persistence>
```

The persistence unit is the bridge between the persistence context and the database. On the one hand, the `<class>` tag lists all the entities that could be managed in the persistence context and, on the other, it defines all the information to physically connect to the database (using properties). The persistence unit also defines the transaction mode. In an application-managed environment, the transaction mode is local (`transaction-type="RESOURCE_LOCAL"`). Chapter 9 will cover container-managed environments (e.g. Spring and Java EE). In container-managed environments, the `persistence.xml` would define a data source instead of the database connection properties and set the transaction type to JTA (`transaction-type="JTA"`) or *Java Transaction API*.

From JPA 2.1 onwards, some properties of the `persistence.xml` file have been standardised (see Table 1). They all start with `javax.persistence` such as `javax.persistence.jdbc.url`. JPA providers are required to support these standard properties, but they may provide custom properties of their own (e.g. the EclipseLink property `eclipselink.logging.level` to setup the log level).

Table 1. Standard persistence.xml Properties

Property	Description
`javax.persistence.jdbc.driver`	Fully qualified name of the driver class
`javax.persistence.jdbc.url`	Driver-specific URL
`javax.persistence.jdbc.user`	Username used by database connection
`javax.persistence.jdbc.password`	Password used by database connection
`javax.persistence.database-product-name`	Name of the targeted database (e.g. H2)
`javax.persistence.database-major-version`	Version number of the targeted database
`javax.persistence.database-minor-version`	Minor version number of the targeted database
`javax.persistence.ddl-create-script-source`	Name of the script creating the database
`javax.persistence.ddl-drop-script-source`	Name of the script dropping the database
`javax.persistence.sql-load-script-`	Name of the script loading data into

`source`	the database
`javax.persistence.schema-generation.database.action`	Specifies the action to be taken with regard to the database (`none`, `create`, `drop-and-create`, `drop`)
`javax.persistence.schema-generation.scripts.action`	Specifies the action to be taken with regard to DDL scripts (`none`, `create`, `drop-and-create`, `drop`)
`javax.persistence.schema-generation.scripts.create-target`	If scripts are to be generated, name of the create script
`javax.persistence.schema-generation.scripts.drop-target`	If scripts are to be generated, name of the drop script
`javax.persistence.lock.timeout`	Value in milliseconds for pessimistic lock timeout
`javax.persistence.query.timeout`	Value in milliseconds for query timeout
`javax.persistence.validation.group.pre-persist`	Groups targeted for validation upon pre-persist
`javax.persistence.validation.group.pre-update`	Groups targeted for validation upon pre-update
`javax.persistence.validation.group.pre-remove`	Groups targeted for validation upon pre-remove

Manipulating Entities

Being the central piece of JPA, we use the entity manager for both simple entity manipulation and complex JPQL query execution. When manipulating single entities, the `EntityManager` interface can be seen as a generic *Data Access Object* (DAO), which allows CRUD operations on any entity (see Listing 1).[1]

To help you gain a better understanding of these methods, I use a simple example of a one-way, one-to-one relationship between a `Customer` and an `Address`. Both entities have automatically generated identifiers (thanks to the `@GeneratedValue` annotation), and `Customer` (see Listing 5) has a lazy fetch to `Address` (see Listing 6).

Listing 5. The Customer Entity with a One-way, One-to-one Address

```
@Entity
public class Customer {

  @Id
  @GeneratedValue
  private Long id;
  private String firstName;
  private String lastName;
  private String email;
  @OneToOne(fetch = LAZY)
  @JoinColumn(name = "address_fk")
  private Address address;

  // Constructors, getters, setters
}
```

Listing 6. The Address Entity

```
@Entity
public class Address {

  @Id
  @GeneratedValue
  private Long id;
  private String street1;
  private String city;
  private String zipcode;
  private String country;

  // Constructors, getters, setters
}
```

These two entities will get mapped into the database structure shown in Figure 3. Note the `ADDRESS_FK` column is the foreign key to `ADDRESS`.

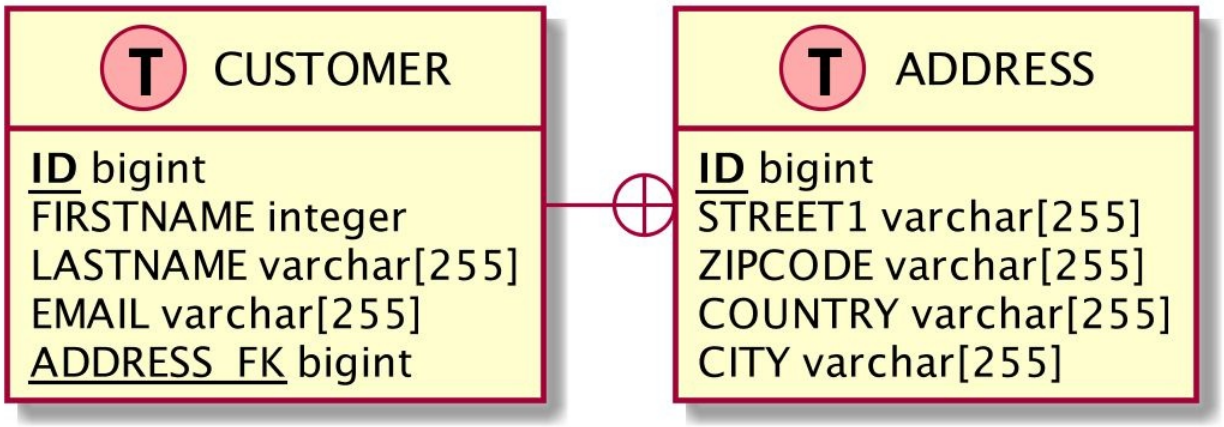

Figure 3. CUSTOMER and ADDRESS tables

For better readability, the fragments of code used in the upcoming sections assume that the `em` attribute is of type `EntityManager` and `tx` of type `EntityTransaction`.

Persisting an Entity

Persisting an entity means inserting data into the database when the data doesn't already exist. To do so, it's necessary to create a new entity instance using the `new` operator, set the values of the attributes, bind one entity to another when there are associations (`customer.setAddress(address)`), and finally call the `EntityManager.persist()` method as shown in the JUnit test case in Listing 7.

Listing 7. Persisting a Customer with an Address

```
Customer customer = new Customer("Anthony", "Balla",
"aballa@mail.com");
Address address = new Address("Ritherdon Rd", "London", "8QE",
"UK");
customer.setAddress(address);

// Persists the object
tx.begin();
em.persist(customer);
em.persist(address);
tx.commit();

assertNotNull(customer.getId());
assertNotNull(address.getId());
```

In Listing 7, `customer` and `address` are just two objects that reside in the JVM memory. Both become managed entities when the entity manager (variable `em`) takes them into account by persisting them (`em.persist()`). At this time, both objects become eligible to be inserted in the database. When the transaction is committed (`tx.commit()`), the data is flushed to the database, an address row is inserted into the `ADDRESS` table, and a customer row is inserted into the `CUSTOMER` table. As the `Customer` is the owner of the

relationship, its table holds the foreign key to `ADDRESS`. The `assertNotNull` expressions check that both entities have received a generated identifier (thanks to the persistence provider and the `@Id` and `@GeneratedValue` annotations).

Note the ordering of the `persist()` methods: a customer is persisted and then an address. If it were the other way round, the result would be the same. Earlier, the entity manager was described as a first-level cache. Until the transaction is committed, the data stays in memory and there is no access to the database. The entity manager caches data and, when the transaction is committed, flushes the data in the order that the underlying database is expecting (respecting integrity constraints). Because of the foreign key in the `CUSTOMER` table, the `insert` statement for `ADDRESS` will be executed first, followed by that for `CUSTOMER`.

Most of the entities in this chapter do not implement the `Serializable` interface. That's because entities don't have to be in order to get persisted in the database. They are passed by reference from one method to the other, and, when they have to be persisted, the `EntityManager.persist()` method is invoked. But, if you need to pass entities by value (remote invocation, external container etc.), they must implement the `java.io.Serializable` marker (no method) interface. It indicates to the compiler that it must enforce all fields on the entity class to be serialisable, so that any instance can be serialised to a byte stream and passed using *Remote Method Invocation* (RMI).

Finding by Id

To find an entity by its identifier, you can use two different methods. The first is the `EntityManager.find()` method, which has two parameters: the entity class and the unique identifier (see Listing 8). If the entity is found, it is returned; if it is not found, a null value is returned.

Listing 8. Finding a Customer by Id

```
Customer customer = em.find(Customer.class, id);
if (customer != null) {
  // Process the object
}
```

The second method is `getReference()` (see Listing 9). It is very similar to the `find` operation, as it has the same parameters, but it retrieves a reference to an entity (via its primary key) but does not retrieve its data. Think of it as a proxy to an entity, not the entity itself. It is intended for situations where a managed entity instance is needed, but no data, other than potentially the entity's primary key, being accessed. With `getReference()`, the state data is fetched lazily, which means that if you don't access state before the entity is detached, the data might not be there. If the entity is not found, an `EntityNotFoundException` is thrown.

Listing 9. Finding a Customer by Reference

```
try {
  Customer customer = em.getReference(Customer.class, id);
  // Process the object
  assertNotNull(customer);
} catch (
  EntityNotFoundException ex) {
  // Entity not found
}
```

Removing an Entity

An entity can be removed with the `EntityManager.remove()` method. Once removed, the entity is deleted from the database, is detached from the entity manager, and cannot be synchronised with the database anymore. In terms of Java objects, the entity is still accessible until it goes out of scope and the garbage collector cleans it up. The code in Listing 10 shows how to remove an object after it has been created.

Listing 10. Creating and Removing Customer and Address Entities

```
Customer customer = new Customer("Anthony", "Balla",
"aballa@mail.com");
Address address = new Address("Ritherdon Rd", "London", "8QE",
"UK");
customer.setAddress(address);

// Persists the object
tx.begin();
em.persist(customer);
em.persist(address);
tx.commit();

assertNotNull(customer.getId());
assertNotNull(address.getId());

// Removes the object from the database
tx.begin();
em.remove(customer);
tx.commit();

// The objects are still available until GC
assertEquals(customer.getFirstName(), "Anthony");
assertEquals(address.getCity(), "London");

// The entities are not in the database
customer = em.find(Customer.class, customer.getId());
```

```
assertNull(customer);
address = em.find(Address.class, address.getId());
assertNotNull(address);
```

The code in Listing 10 creates an instance of `Customer` and `Address`, links them together (`customer.setAddress(address)`), and persists them. In the database, the customer row is linked to the address through a foreign key; later on, in the code, only the `Customer` is deleted. Depending on how the cascading is configured (discussed later in this chapter), the `Address` could be left with no other entity referencing it and the address row becomes an orphan.

Orphan Removal

For data consistency, orphans are not desirable, as they result in having rows that are not referenced by any other table in the database, without means of access. With JPA, you can inform the persistence provider to automatically remove orphans or cascade a remove operation as you'll see later. If a target entity (`Address`) is privately owned by a source (`Customer`), meaning a target must never be owned by more than one source, and that source is deleted by the application, the provider should also delete the target.

Associations that are specified as one-to-one, or one-to-many, support the use of the orphan-removal option. To include this option in the example, we just add the `orphanRemoval=true` element to the `@OneToOne` annotation (see Listing 11).

Listing 11. The Customer Entity Dealing with Orphan Address Removal

```
@Entity
public class Customer {

  @Id
  @GeneratedValue
  private Long id;
  private String firstName;
  private String lastName;
  private String email;
  @OneToOne(orphanRemoval = true)
  @JoinColumn(name = "address_fk")
  private Address address;

  // Constructors, getters, setters
}
```

With this mapping, the code in Listing 12 will automatically remove the `Address` entity when the customer is removed, or when the relationship is broken (by setting the `address` attribute to null, or by removing the child entity from the collection in a one-to-many case). The remove operation is applied at the time of the flush operation (transaction committed).

Listing 12. Removing Only the Customer Entity

```
Customer customer = new Customer("Anthony", "Balla",
"tballa@mail.com");
Address address = new Address("Ritherdon Rd", "London", "8QE",
"UK");
customer.setAddress(address);

// Persists the object
tx.begin();
em.persist(customer);
em.persist(address);
tx.commit();

assertNotNull(customer.getId());
assertNotNull(address.getId());
```

```
// Removes only the customer entity from the database
tx.begin();
em.remove(customer);
tx.commit();

// Customer is not in the database, nor address
customer = em.find(Customer.class, customer.getId());
assertNull(customer);
address = em.find(Address.class, address.getId());
assertNull(address);
```

Synchronising with the Database

Up to now, the synchronisation with the database has been done at commit time. The entity manager is a first-level cache, waiting for the transaction to be committed in order to flush the data to the database, but what happens when a customer and an address need to be inserted?

```
tx.begin();
em.persist(customer);
em.persist(address);
tx.commit();
```

All pending changes require an SQL statement; here, two `insert` statements are produced and made permanent only when the database transaction commits. For most applications, this automatic data synchronisation is sufficient. The database is synchronised with the entities in the persistence context, but data can be explicitly flushed (`flush`) to the database, or entities refreshed with data from the database (`refresh`).

If the data is flushed to the database at one point, and if, later in the code, the application calls the `rollback()` method, the flushed data will be taken out of the database.

Flushing an Entity

With the `EntityManager.flush()` method, the persistence provider can be explicitly forced to flush the data to the database but it won't commit the transaction. This allows a developer to manually trigger the same process used by the entity manager internally to flush the persistence context.

Listing 13. Flushing the Customer Entity to the Database

```
Customer customer = new Customer("Anthony", "Balla",
"aballa@mail.com");
Address address = new Address("Ritherdon Rd", "London", "8QE",
"UK");
customer.setAddress(address);

assertThrows(IllegalStateException.class, () -> {
  tx.begin();
  em.persist(customer);
  em.flush();
  em.persist(address);
  tx.commit();
});
```

Two interesting things happen in Listing 13. The first is that `em.flush()` will not wait for the transaction to commit and will force the provider to flush the persistence context. An insert statement will be generated and executed at the flush. The second is that this code will not work because of the integrity constraint. Without an explicit flush, the entity manager caches all changes and

orders and executes them in a coherent way for the database. With an explicit flush, the insert statement to `CUSTOMER` will be executed, but the state of the `Address` entity is not consistent. That's because the `Address` entity is not persisted yet, therefore, it doesn't have an id (the id is set to null) but all the other attributes are set (street, city etc.). That's why we get an `IllegalStateException`: the `Customer` entity has an `Address` with set values except for the identifier. That will lead the transaction to roll back. Data that has been flushed will also get rolled back. Explicit flushes should be carefully used and only when needed.

Refreshing an Entity

The `refresh()` method is used for data synchronisation in the opposite direction of the flush, meaning it overwrites the current state of a managed entity with the data present in the database. A typical case is when you use the `EntityManager.refresh()` method to undo changes that have been made to the entity in memory only. The test case snippet in Listing 14 finds a `Customer` by Id, changes its first name, and undoes this change using the `refresh()` method.

Listing 14. Refreshing the Customer Entity from the Database

```
Customer customer = em.find(Customer.class, id);
assertEquals("Anthony", customer.getFirstName());

customer.setFirstName("William");
assertEquals("William", customer.getFirstName());
```

```
// Refreshes the customer entity
em.refresh(customer);
assertEquals("Anthony", customer.getFirstName());
```

Content of the Persistence Context

The persistence context holds the managed entities. With the `EntityManager` interface, you can check whether an entity is being managed, detach it, or clear all entities from the persistence context.

Contains

Entities are either managed or not by the entity manager. The `EntityManager.contains()` method returns a `Boolean` and allows you to check whether or not a particular entity instance is currently managed by the entity manager within the current persistence context. In the test case in Listing 15, a `Customer` is persisted, and you can immediately check whether the entity is managed (`em.contains(customer)`). The answer is true. Afterwards, the `remove()` method is called, and the entity is removed from the database and from the persistence context (`em.contains(customer)` returns `false`).

Listing 15. Test Case for Whether the Customer Entity Is in the Persistence Context

```
Customer customer = new Customer("Anthony", "Balla",
"aballa@mail.com");

assertFalse(em.contains(customer));

// Persists the object
tx.begin();
em.persist(customer);
```

```
tx.commit();

assertTrue(em.contains(customer));

// Removes the object
tx.begin();
em.remove(customer);
tx.commit();

assertFalse(em.contains(customer));
```

Clear and Detach

The `clear()` method is straightforward: it empties the persistence context, causing all managed entities to become detached (check the "Entity Life Cycle" introduced in Chapter 3, more on that in Chapter 7). The `detach(Object entity)` method removes the given entity from the persistence context. Changes made to the entity will not be synchronised to the database after such an eviction has taken place. Listing 16 creates an entity, checks that it is managed, detaches it from the persistence context, and checks that it is detached.

Listing 16. Checking Whether the Customer Entity Is in the Persistence Context

```
Customer customer = new Customer("Anthony", "Balla",
"aballa@mail.com");

// Persists the object
tx.begin();
em.persist(customer);
tx.commit();

assertTrue(em.contains(customer));

// Detaches the entity
em.detach(customer);

assertFalse(em.contains(customer));
```

Merging an Entity

A detached entity is no longer associated with a persistence context. If you want to manage it, you need to reattach it (i.e. merge it). Let's take the example of an entity that needs to be displayed on a web page. The entity is first loaded from the database into the persistent layer (it is managed), it is returned from an invocation of a transactional bean (it is detached because the transaction context ends), the presentation layer displays it (it is still detached), and then it returns to be updated to the database. However, at that moment, the entity is detached and needs to be attached again, in order to synchronise its state with the database. This reattachment happens through the `em.merge()` method.

Listing 17 simulates this case by clearing the persistence context (`em.clear()`), which detaches the entity.

Listing 17. Clearing the Persistence Context and Merging an Entity

```
Customer customer = new Customer("Anthony", "Balla",
"aballa@mail.com");

// Persists the object
tx.begin();
em.persist(customer);
tx.commit();
assertTrue(em.contains(customer));

// Clears the Persistence Context
em.clear();
assertFalse(em.contains(customer));

// Re-attaches the entity and updates name
customer.setFirstName("William");
tx.begin();
customer = em.merge(customer);
```

```
tx.commit();
assertTrue(em.contains(customer));

// Clears the Persistence Context
em.clear();
assertFalse(em.contains(customer));

// Checks the name has been updated
customer = em.find(Customer.class, customer.getId());
assertEquals(customer.getFirstName(), "William");
assertTrue(em.contains(customer));
```

The code in Listing 17 creates and persists a customer. The call to `em.clear()` forces the detachment of the customer entity, but detached entities continue to live outside the persistence context in which they were located, and their state is no longer guaranteed to be synchronised with the database state. That's what happens with `customer.setFirstName("William")`. This is executed on a detached entity, and the data is not updated in the database. To synchronise this change with the database, you need to reattach the entity (i.e. merge it) with `em.merge(customer)` inside a transaction.

Updating an Entity

Updating an entity is simple, yet at the same time it can be confusing to understand. As you've just seen, you can use `EntityManager.merge()` to attach an entity and synchronise its state with the database. But, if an entity is currently managed, changes to it will be reflected in the database automatically (after the transaction is committed). If not, you will need to explicitly call `merge()`.

Listing 18 demonstrates persisting a customer with a first name set to `Anthony`. When you call the `em.persist()` method, the entity is managed, so any changes made to the entity will be synchronised with the database. When you call the `setFirstName()` method , the entity changes its state. The entity manager caches any entity action starting at `tx.begin()` and synchronises it when committed.

Listing 18. Updating the Customer's First Name

```
Customer customer = new Customer("Anthony", "Balla",
"aballa@mail.com");

// Persists the object
tx.begin();
em.persist(customer);

assertEquals(customer.getFirstName(), "Anthony");

// Updates name while managed
customer.setFirstName("William");
assertEquals(customer.getFirstName(), "William");

tx.commit();

customer = em.find(Customer.class, customer.getId());
assertEquals(customer.getFirstName(), "William");
```

Cascading Events

By default, every entity manager operation applies only to the entity supplied as an argument to the operation. But sometimes, when an operation is carried out on an entity, you want to propagate the operation on the entity's associations. This is known as *cascading an event*. The examples so far have relied on default cascade behaviour and not customised behaviour. In Listing 19, to create a customer,

you instantiate a `Customer` and an `Address` entity, link them together (`customer.setAddress(address)`), and then persist the two.

Listing 19. Persisting a Customer with an Address

```
Customer customer = new Customer("Anthony", "Balla",
"aballa@mail.com");
Address address = new Address("Ritherdon Rd", "London", "8QE",
"UK");
customer.setAddress(address);

// Persists the object
tx.begin();
em.persist(customer);
em.persist(address);
tx.commit();

assertNotNull(customer.getId());
assertNotNull(address.getId());
```

Because there's a relationship between `Customer` and `Address`, you could cascade the `persist` action from the customer to the address. That would mean that a call to `em.persist(customer)` would cascade the `persist` event to the `Address` entity if it allows this type of event to be propagated. You could then shrink the code and do away with the `em.persist(address)` as shown in Listing 20.

Listing 20. Cascading a Persist Event to Address

```
Customer customer = new Customer("Anthony", "Balla",
"aballa@mail.com");
Address address = new Address("Ritherdon Rd", "London", "8QE",
"UK");
customer.setAddress(address);

// Persists the object
```

```
tx.begin();
em.persist(customer);
tx.commit();

assertNotNull(customer.getId());
assertNotNull(address.getId());
```

Without cascading, the customer would get persisted but not the address. Cascading an event is possible if the mapping of the relationship is modified. The annotations `@OneToOne`, `@OneToMany`, `@ManyToOne`, and `@ManyToMany` have a `cascade` attribute that takes an array of events to be cascaded. To allow the cascading of the `persist`, you must modify the mapping of the `Customer` entity (see Listing 21) and add a `cascade` attribute with a `PERSIST` event (as well as a `REMOVE` event which is commonly used to perform delete cascades).

Listing 21. Customer Entity Cascading Persist and Remove Events

```
@Entity
public class Customer {

  @Id
  @GeneratedValue
  private Long id;
  private String firstName;
  private String lastName;
  private String email;
  @OneToOne(fetch = LAZY, cascade = {PERSIST, REMOVE})
  @JoinColumn(name = "address_fk")
  private Address address;

  // Constructors, getters, setters
}
```

You can choose from several events to cascade to a target association (Table 2 lists these events) and you can even cascade them all using the `CascadeType.ALL` type.

Table 2. Possible Events to Be Cascaded

Property	Description
`PERSIST`	Cascades persist operations to the target of the association
`REMOVE`	Cascades remove operations to the target of the association
`MERGE`	Cascades merge operations to the target of the association
`REFRESH`	Cascades refresh operations to the target of the association
`DETACH`	Cascades detach operations to the target of the association
`ALL`	Declares that all the previous operations should be cascaded

But remember that, by default, no event is cascaded from the parent to the child entities.

1 DAO https://en.wikipedia.org/wiki/Data_access_object