Pro EJB 3

Java Persistence API

Mike Keith

Merrick Schincariol

Apress®

Pro EJB 3: Java Persistence API

ISBN-13 (pbk): 978-1-59059-645-6

ISBN-10 (pbk): 1-59059-645-5

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Steve Anglin
Technical Reviewer: Jason Haley, Huyen Nguyen, Shahid Shah
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Manager: Julie M. Smith
Copy Edit Manager: Nicole LeClerc
Copy Editor: Hastings Hart
Assistant Production Director: Kari Brooks-Copony
Production Editor: Laura Esterman
Compositors: Pat Christenson and Susan Glinert Stevens
Proofreader: Elizabeth Berry
Indexer: Julie Grady
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section. You will need to answer questions pertaining to this book in order to successfully download the code.

Contents at a Glance

Contents

CHAPTER 2

Getting Started

From the outset, one of the main goals when creating the Java Persistence API was to ensure that it is simple to use and easy to understand. Although the problem domain cannot be trivialized or watered down, the technology that enables one to deal with it can be straightforward and intuitive. In this chapter we will show how effortless it is to develop and use entities.

We will start this chapter off by describing the basic characteristics of entities. We'll define what an entity is and how to create, read, update, and delete them. We'll also introduce entity managers and how they are obtained and used. Then we'll take a quick look at queries and how to specify and execute a query using the `EntityManager` and `Query` objects. The chapter will conclude by showing a simple working application that runs in a standard Java SE 5 environment and that demonstrates all of the example code in action.

Entity Overview

The entity is not a new thing. In fact, entities have been around longer than many programming languages and certainly longer than Java. They were first introduced by Peter Chen in his seminal paper on entity-relationship modeling.[1] He described entities as things that have attributes and relationships. The expectation was that the attributes were going to be persisted in a relational database, as were the relationships.

Even now, the definition still holds true. An *entity* is essentially a noun, or a grouping of state associated together as a single unit. It may participate in relationships to any number of other entities in a number of standard ways. In the object-oriented paradigm, we would add behavior to it and call it an object. In the Java Persistence API, any application-defined object can be an entity, so the important question might be, What are the characteristics of an object that has been turned into an entity?

Persistability

The first and most basic characteristic of entities is that they are *persistable*. This generally just means that they can be made persistent. More specifically it means that their state can be represented in a data store and can be accessed at a later time, perhaps well after the end of the process that created it.

1. Peter C. Chen, "The entity-relationship model—toward a unified view of data," *ACM Transactions on Database Systems* 1, no. 1 (1976): 9–36.

We could call them persistent objects, and many people do, but it is not technically correct. Strictly speaking, a persistent object becomes persistent the moment it is instantiated. If a persistent object exists, then by definition it is already persistent.

An entity is persistable because it *can* be created in a persistent store. The difference is that it is not automatically persisted and that in order for it to have a persistent representation the application must actively invoke an API method to initiate the process. This is an important distinction because it leaves control over persistence firmly in the hands of the application. It offers the application the flexibility to manipulate data and perform business logic on the entity, and then only when the application decides that it is the right time to persist the entity, actually causing it to be persistent. The lesson is that entities may be manipulated without necessarily having persistent repercussions, and it is the application that decides whether or not they do.

Identity

Like any other Java object, an entity has an object identity, but when it exists in the data store it also has a *persistent identity*. Persistent identity, or an *identifier*, is the key that uniquely identifies an entity instance and distinguishes it from all of the other instances of the same entity type. An entity has a persistent identity when there exists a representation of it in the data store, that is, a row in a database table. If it is not in the database then even though the in-memory entity may have its identity set in a field, it does not have a persistent identity. The entity identifier, then, is equivalent to the primary key in the database table that stores the entity state.

Transactionality

Entities are what we might call *quasi-transactional*. They are normally only created, updated, and deleted within a transaction,[2] and a transaction is required for the changes to be committed in the database. Changes made to the database either succeed or fail atomically, so the persistent view of an entity should indeed be transactional.

In memory it is a slightly different story in the sense that entities may be changed without the changes ever being persisted. Even when enlisted in a transaction, they may be left in an undefined or inconsistent state in the event of a rollback or transaction failure. The in-memory entities are simple Java objects that obey all of the rules and constraints that are applied by the Java virtual machine to other Java objects.

Granularity

Finally, we can also learn something about what entities are by describing what they are *not*. They are not primitives, primitive wrappers, or built-in objects. These are no more than scalars and do not have any designated semantic meaning to an application. A string, for example is too fine-grained an object to be an entity because it does not have any domain-specific connotation. Rather, a string is well-suited and very often used as a type for an entity attribute and given meaning according to the entity attribute that it is typing.

2. In most cases this is a requirement, but in certain configurations the transaction may not be present until later.

Entities are fine-grained objects that have a set of aggregated state that is normally stored in a single place, such as a row in a table, and typically have relationships to other entities. In the most general sense they are business domain objects that have specific meaning to the application that accesses them.

While it is certainly true that entities may be defined in exaggerated ways to be as fine-grained as storing a single string or coarse-grained enough to contain 500 columns' worth of data, the suggested granularity of an entity is definitely on the smaller end of the spectrum. Ideally, entities should be designed and defined as fairly lightweight objects of equal or smaller size than that of the average Java object.

Entity Metadata

Associated with every entity is metadata in some amount, possibly small, that describes it. This metadata enables the persistence layer to recognize, interpret, and properly manage the entity from the time it is loaded through to its runtime invocation.

The metadata that is actually required for each entity is minimal, rendering entities easy to define and use. However, like any sophisticated technology with its share of switches, levers, and buttons, there is also the possibility to specify much, much more metadata than is required. It may be extensive amounts, depending upon the application requirements, and may be used to customize every detail of the entity configuration or state mappings.

Entity metadata may be specified in one of two ways—annotations or XML. Each is equally valid, but the one that you use will depend upon your development preferences or process.

Annotations

Annotation metadata is a language feature that allows structured and typed metadata to be attached to the source code. It was introduced as part of Java SE 5 and is a key part of the EJB 3.0 and Java EE 5 specifications.[3] Although annotations are not required by the Java Persistence API, they are a convenient way to learn and use the API. Because annotations co-locate the metadata with the program artifacts, it is not necessary to escape to an additional file and additional language (XML) just to specify the metadata.

Annotations are used throughout both the examples and the accompanying explanations in this book. All of the API annotations that are shown and described, except for Chapter 3, which talks about Java EE annotations, are defined in the `javax.persistence` package. Example code snippets can be assumed to have an implicit import of the form `import javax.persistence.*;`.

XML

For those who prefer to use the traditional XML descriptors, this option is still available. It should be a fairly straightforward process to switch to using XML descriptors after having learned and understood the annotations since the XML has in large part been patterned after the annotations. Chapter 10 describes how to use XML to specify or override entity mapping metadata.

3. The Java EE 5 platform specification and all of its sub-specifications require the use of Java SE 5.

ANNOTATIONS

Java annotations are specially defined types that may annotate (be attached to or placed in front of) Java programming elements including classes, methods, fields, and variables. When they annotate a program element, the compiler reads the information contained in them and may retain it in the class files or dispose of it according to what was specified in the annotation type definition. When retained in the class files the elements contained in the annotation may be queried at runtime through a reflection-based API. A running program can in this way obtain the metadata that exists on a Java program element. An example of a custom annotation type definition that could be used to indicate classes that should be validated (whatever validate means to the application or tool that is processing it) is:

```
@Target(TYPE) @Retention(RUNTIME)
public @interface Validate {
    boolean flag;
}
```

This annotation definition is in fact itself annotated by `@Target` and `@Retention` built-in annotations that determine what kinds of program elements the annotation may annotate and at what point the annotation metadata should be discarded from the class. The annotation defined above may annotate any type and will not be discarded from the class (that is, it will be retained in the class file even at runtime). This annotation may, for example, annotate any given class definition. An example usage of this annotation could be:

```
@Validate(flag=true)
public class MyClass {
  ...
}
```

An application that looks at all classes in the system for this annotation will be able to determine that `MyClass` should be validated and perform that validation whenever it makes sense. The semantic meaning of `@Validate` is completely up to the component that defines the annotation type and the one that reads and processes the annotation.

Configuration by Exception

The notion of *configuration by exception* means that the persistence engine defines defaults that apply to the majority of applications and that users need to supply values only when they want to override the default value. In other words, having to supply a configuration value is the exception to the rule, not a requirement.

Configuration by exception is ingrained in the Java Persistence API and is a strong contributing factor to its usability. The majority of configuration values have defaults, rendering the metadata that does have to be specified more relevant and concise.

The extensive use of defaults and the ease of use that it brings to configuration comes with a price, however. When defaults are embedded into the API and do not have to be specified, then they are not visible or obvious to users. This *can* make it possible for users to be unaware of the complexity of developing persistence applications, making it slightly more difficult to debug or to change the behavior when it becomes necessary.

Defaults are not meant to shield users from the often complex issues surrounding persistence. They are meant to allow a developer to get started easily and quickly with something that will work and then iteratively improve and implement additional functionality as the complexity of their application increases. Even though the defaults may be what you want to have happen most of the time, it is still fairly important for developers to be familiar with the default values that are being applied. For example, if a table name default is being assumed, then it is important to know what table the runtime is expecting, or if schema generation is used, what table will be generated.

For each of the annotations we will also discuss the default value so that it is clear what will be applied if the annotation is not specified. We recommend that you remember these defaults as you learn them. After all, a default value is still part of the configuration of the application; it was just really easy to configure!

Creating an Entity

Regular Java classes are easily transformed into entities simply by annotating them. In fact, by adding a couple of annotations, virtually any class with a no-arg constructor can become an entity.

Let's start by creating a regular Java class for an employee. Listing 2-1 shows a simple `Employee` class.

Listing 2-1. `Employee` *Class*

```
public class Employee {
    private int id;
    private String name;
    private long salary;

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

    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public long getSalary() { return salary; }
    public void setSalary (long salary) { this.salary = salary; }
}
```

You may notice that this class resembles a JavaBean-style class with three properties: `id`, `name`, and `salary`. Each of these properties is represented by a pair of accessor methods to get and set the property and is backed by a member field. Properties or member fields are the units of state within the entity that we want to persist.

To turn `Employee` into an entity we first need to annotate the class with `@Entity`. This is primarily just a marker annotation to indicate to the persistence engine that the class is an entity.

The second annotation that we need to add is `@Id`. This annotates the particular field or property that holds the persistent identity of the entity (the primary key) and is needed so the provider knows which field or property to use as the unique identifying key in the table.

Adding these two annotations to our `Employee` class, we end up with pretty much the same class that we had before, except that now it is an entity. Listing 2-2 shows the entity class.

Listing 2-2. `Employee` *Entity*

```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;

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

    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public long getSalary() { return salary; }
    public void setSalary (long salary) { this.salary = salary; }
}
```

When we say that the `@Id` annotation is placed on the field or property, we mean that the user can choose to annotate either the declared field, or the getter method[4] of a JavaBean-style property. Either field or property strategy is allowed, depending upon the needs and tastes of the entity developer, but whichever strategy is chosen, it must be followed for all persistent state annotations in the entity. We have chosen in this example to annotate the field because it is simpler; in general, this will be the easiest and most direct approach. We will learn more about the details of annotating persistent state using field or property access in subsequent chapters.

Automatic State Mapping

The fields in the entity are automatically made persistable by virtue of their existence in the entity. Default mapping and loading configuration values apply to these fields and enable them to be persisted when the object is persisted. Given the questions that were brought up in the last chapter, one might be led to ask, "How did the fields get mapped, and where do they get persisted to?"

To find the answer we must first take a quick detour to dig inside the `@Entity` annotation and look at an element called `name` that uniquely identifies the entity. The entity name may be explicitly specified for any entity by using this `name` element in the annotation, as in `@Entity(name="Emp")`. In practice this is seldom specified because it gets defaulted to be the unqualified name of the entity class. This is almost always both reasonable and adequate.

Now we can get back to the question about where the data gets stored. It turns out that the default name of the table used to store any given entity of a particular entity type is the name

4. Annotations on setter methods will just be ignored.

of the entity. If we have specified the name of the entity, then that will be the default table name, but if we have not, then the default value of the entity name will be used. We just stated that the default entity name was the unqualified name of the entity class, so that is effectively the answer to the question of which table gets used. In our `Employee` example all entities of type `Employee` will get stored in a table called `EMPLOYEE`.

Each of the fields or properties has individual state in it and needs to be directed to a particular column in the table. We know to go to the `EMPLOYEE` table, but we don't know which column to use for any given field or property. When no columns are explicitly specified, then the default column is used for a field or property, which is just the name of the field or property itself. So our employee id will get stored in the `ID` column, the name in the `NAME` column, and the salary in the `SALARY` column of the `EMPLOYEE` table.

Of course these values can all be overridden to match an existing schema. We will discuss how to override them when we get to Chapter 4 and discuss mapping in more detail.

Entity Manager

In the Entity Overview section, it was stated that a specific API call needs to be invoked before an entity actually gets persisted to the database. In fact, separate API calls are needed to perform many of the operations on entities. This API is implemented by the entity manager and encapsulated almost entirely within a single interface called `EntityManager`. When all is said and done, it is to an entity manager that the real work of persistence is delegated. Until an entity manager is used to actually create, read, or write an entity, the entity is nothing more than a regular (non-persistent) Java object.

When an entity manager obtains a reference to an entity, either by having it explicitly passed in or because it was read from the database, that object is said to be *managed* by the entity manager. The set of managed entity instances within an entity manager at any given time is called its *persistence context*. Only one Java instance with the same persistent identity may exist in a persistence context at any time. For example, if an `Employee` with a persistent identity (or id) of 158 exists in the persistence context, then no other object with its id set to 158 may exist within that same persistence context.

Entity managers are configured to be able to persist or manage specific types of objects, read and write to a given database, and be implemented by a particular *persistence provider* (or *provider* for short). It is the provider that supplies the backing implementation engine for the entire Java Persistence API, from the `EntityManager` through to `Query` implementation and SQL generation.

All entity managers come from factories of type `EntityManagerFactory`. The configuration for an entity manager is bound to the `EntityManagerFactory` that created it, but it is defined separately as a *persistence unit*. A persistence unit dictates either implicitly or explicitly the settings and entity classes used by all entity managers obtained from the unique `EntityManagerFactory` instance bound to that persistence unit. There is, therefore, a one-to-one correspondence between a persistence unit and its concrete `EntityManagerFactory`.

Persistence units are named to allow differentiation of one `EntityManagerFactory` from another. This gives the application control over which configuration or persistence unit is to be used for operating on a particular entity.

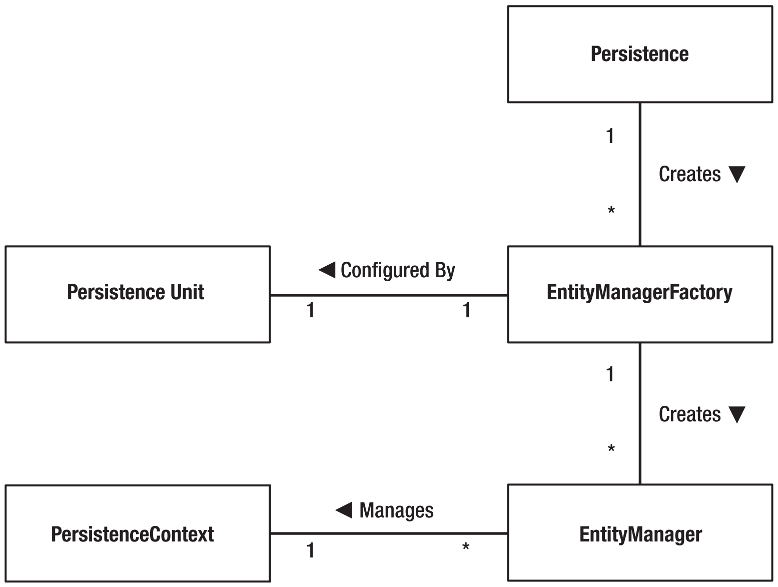

Figure 2-1. *Relationships between Java Persistence API concepts*

Figure 2-1 shows that for each persistence unit there is an `EntityManagerFactory` and that many entity managers can be created from a single `EntityManagerFactory`. The part that may come as a surprise is that many entity managers can point to the same persistence context. We have talked only about an entity manager and its persistence context, but later on we will see that this is indeed the case and that there may be multiple references to different entity managers which all point to the same group of managed entities.

Obtaining an Entity Manager

An entity manager is always obtained from an `EntityManagerFactory`. The factory from which it was obtained determines the configuration parameters that govern its operation. While there are shortcuts that veil the factory from the user view when running in a Java EE application server environment, in the Java SE environment we can use a simple bootstrap class called `Persistence`. The static `createEntityManagerFactory()` method in the `Persistence` class returns the `EntityManagerFactory` for the specified persistence unit name. The following example demonstrates creating an `EntityManagerFactory` for the persistence unit named "EmployeeService":

```
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("EmployeeService");
```

The name of the specified persistence unit "EmployeeService" passed into the `createEntityManagerFactory()` method identifies the given persistence unit configuration that determines such things as the connection parameters that entity managers generated from this factory will use when connecting to the database.

Now that we have a factory, we can easily obtain an entity manager from it. The following example demonstrates creating an entity manager from the factory that we acquired in the previous example:

```
EntityManager em = emf.createEntityManager();
```

With this entity manager, we are now in a position to start working with persistent entities.

Persisting an Entity

Persisting an entity is the operation of taking a transient entity, or one that does not yet have any persistent representation in the database, and storing its state so that it can be retrieved later. This is really the basis of persistence—creating state that may outlive the process that created it. We are going to start by using the entity manager to persist an instance of `Employee`. Here is a code example that does just that:

```
Employee emp = new Employee(158);
em.persist(emp);
```

The first line in this code segment is simply creating an `Employee` instance that we want to persist. If we ignore the sad fact that we seem to be employing a nameless individual and paying them nothing (we are setting only the id, not the name or salary) the instantiated `Employee` is just a regular Java object.

The next line obtains an entity manager and uses it to persist the entity. Calling `persist()` is all that is required to initiate it being persisted in the database. If the entity manager encounters a problem doing this, then it will throw an unchecked `PersistenceException`; otherwise the employee will be stored in the database. When the `persist()` call returns, `emp` will be a managed entity within the entity manager's persistence context.

Listing 2-3 shows how to incorporate this into a simple method that creates a new employee and persists it to the database.

Listing 2-3. *Method for Creating an Employee*

```
public Employee createEmployee(int id, String name, long salary) {
    Employee emp = new Employee(id);
    emp.setName(name);
    emp.setSalary(salary);
    em.persist(emp);
    return emp;
}
```

This method assumes the existence of an entity manager in the `em` field of the instance and uses it to persist the `Employee`. Note that we do not need to worry about the failure case in this example. It will result in a runtime `PersistenceException` being thrown, which will get propagated up to the caller.

Finding an Entity

Once an entity is in the database, then the next thing one typically wants to do is find it again. In this section we will show how an entity can be found using the entity manager. There is really only one line that we need to show:

```
Employee emp = em.find(Employee.class, 158);
```

We are passing in the class of the entity that is being sought (in this example we are looking for an instance of `Employee`) and the id or primary key that identifies the particular entity (in our case we want to find the entity that we just created). This is all the information needed by the entity manager to find the instance in the database, and when the call completes, the employee that gets returned will be a managed entity, meaning that it will exist in the current persistence context associated with the entity manager.

PARAMETERIZED TYPES

Another of the principal features included in Java SE 5 was the introduction of generics. The abstraction of Java types allowed them to be parameterized and used generically by a class or method. Such classes or methods that make use of type parameterization are called generic types or generic methods. An example of a generic class is one that defines a parameterized type variable in its definition. It could then use that type in the signature of its methods just as does the following generic class:

```
public class Holder<T> {
    T contents;
    public void setContents(T obj) { contents = obj; }
    public T getContents() { return contents; }
}
```

This `Holder` class is parameterized by the `T` type variable making it possible to create an instance that can hold a given type. Why is this better than simply using `Object` everywhere where `T` is used? The reason is because once the type is supplied and the `Holder` is instantiated to be of a given type, then only instances of that type will be allowed to be stored. This makes any given `Holder` instance strongly typed for the type of our choice. For example, we can do the following:

```
Holder<String> stringHolder = new Holder<String>();
stringHolder.setContents("MyOwnString");
Holder<Integer> intHolder = new Holder<Integer>();
intHolder.setContents(100);
String s = stringHolder.getContents();
stringHolder.setContents(101); // compile error
```

We have a `Holder` that stores `String` objects or anything we want, but once we define it then we get the strong compile-time type checking that frees us from having to type-check at runtime. `ClassCastExceptions` can be a thing of the past (well, almost!). As an added bonus, we don't have to cast. The `getContents()` generic method returns precisely the type that was passed to `Holder` as the type parameter, so the compiler can type-check and safely assign as needed.

You may have noticed that there is no cast required to make the return result an Employee object, even though the find() method call can be used for any type of entity. Those who have used Java SE 5 will recognize that this is just because the return type of the find() method is parameterized to return the same class that was passed in, so if Employee was passed as the entity class, then it will also be the return type.

What happens if the object has been deleted or if we supplied the wrong id by accident? In the event that the object was not found, then the find() call simply returns null. We would need to ensure that a null check is performed before the next time the emp variable is used.

The code for a method that looks up and returns the Employee with a given id is now trivial and shown in Listing 2-4.

Listing 2-4. *Method for Finding an Employee*

```
public Employee findEmployee(int id) {
    return em.find(Employee.class, id);
}
```

In the case where no employee exists for the id that is passed in, then the method will return null, since that is what find() will return.

Removing an Entity

Removal of an entity from the database is not as common a thing as some might think. Many applications simply never delete objects, or if they do they just flag the data as being out of date or no longer valid and then just keep it out of sight of clients. We are not talking about that kind of application-level logical removal, where the data is not actually even removed from the database. We are talking about something that results in a DELETE statement being made across one or more tables.

In order to remove an entity, the entity itself must be managed, meaning that it is present in the persistence context. This means that the calling application should have already loaded or accessed the entity and is now issuing a command to remove it. This is not normally a problem given that most often the application will have caused it to become managed as part of the process of determining that this was the object that it wanted to remove.

A simple example for removing an employee is:

```
Employee emp = em.find(Employee.class, 158);
em.remove(emp);
```

In this example we are first finding the entity using the find() call, which returns a managed instance of Employee, and then removing the entity using the remove() call on the entity manager. Of course, we learned in the previous section that if the entity was not found then the find() method will return null. We would get a java.lang.IllegalArgumentException if it turned out that we passed null into the remove() call because we forgot to include a null check before calling remove().

In our application method for removing an employee, we can fix the problem by checking for the existence of the employee before we issue the remove() call, as shown in Listing 2-5.

Listing 2-5. *Method for Removing an Employee*

```
public void removeEmployee(int id) {
    Employee emp = em.find(Employee.class, id);
    if (emp != null) {
        em.remove(emp);
    }
}
```

This method will ensure that the employee with the given id is removed from the database. It will return successfully whether the employee exists or not.

Updating an Entity

An entity may be updated in a few different ways, but for now we will illustrate the most common and simple case. This is the case where we have a managed entity and want to make changes to it. If we do not have a reference to the managed entity, then we must first obtain one using `find()` and then perform our modifying operations on the managed entity. This code adds $1,000 to the salary of the employee with id 158:

```
Employee emp = em.find(Employee.class, 158);
emp.setSalary(emp.getSalary() + 1000);
```

Note the difference between this operation and the others. In this case we are not calling into the entity manager to modify the object but directly on the object itself. For this reason it is important that the entity be a managed instance, otherwise the persistence provider will have no means of detecting the change, and no changes will be made to the persistent representation of the employee.

Our method to raise the salary of a given employee will take the id and amount of the raise, find the employee, and change the salary to the adjusted one. Listing 2-6 demonstrates this approach.

Listing 2-6. *Method for Updating an Employee*

```
public Employee raiseEmployeeSalary(int id, long raise) {
    Employee emp = em.find(Employee.class, id);
    if (emp != null) {
        emp.setSalary(emp.getSalary() + raise);
    }
    return emp;
}
```

If we can't find the employee, then we return `null` so the caller will know that no change could be made. We indicate success by returning the updated employee.

Transactions

The keen reader may have noticed something in the code to this point that was inconsistent with earlier statements made about transactionality when working with entities. There were no transactions in any of the above examples, even though we said that changes to entities must be made persistent using a transaction.

In all the examples except the one that only called `find()`, we assume that a transaction enclosed each method. The `find()` call is not a mutating operation, so it may be called any time, with or without a transaction.

Once again, the key is the environment in which the code is being executed. The typical situation when running inside the Java EE container environment is that the standard Java Transaction API (JTA) is used. The transaction model when running in the container is to assume the application will ensure that a transactional context is present when one is required. If a transaction is not present, then either the modifying operation will throw an exception or the change will simply never be persisted to the data store. We will come back to discussing transactions in the Java EE environment in more detail in Chapter 3.

In our example in this chapter, though, we are not running in Java EE. We are in a Java SE environment, and the transaction service that should be used in Java SE is the `EntityTransaction` service. When executing in Java SE we either need to begin and to commit the transaction in the operational methods, or we need to begin and to commit the transaction before and after calling an operational method. In either case, a transaction is started by calling `getTransaction()` on the entity manager to get the `EntityTransaction` and then invoking `begin()` on it. Likewise, to commit the transaction the `commit()` call is invoked on the `EntityTransaction` obtained from the entity manager. For example, starting and committing before and after the method would produce code that creates an employee the way it is done in Listing 2-7.

Listing 2-7. *Beginning and Committing an* `EntityTransaction`

```
em.getTransaction().begin();
createEmployee(158, "John Doe", 45000);
em.getTransaction().commit();
```

Further detail about resource-level transactions and the `EntityTransaction` API are contained in Chapter 5.

Queries

In general, given that most developers have used a relational database at some point or other in their lives, most of us pretty much know what a database query is. In the Java Persistence API, a query is similar to a database query, except that instead of using Structured Query Language (SQL) to specify the query criteria, we are querying over entities and using a language called Java Persistence Query Language (which we will abbreviate as JPQL).

A query is implemented in code as a `Query` object. `Query` objects are constructed using the `EntityManager` as a factory. The `EntityManager` interface includes a variety of API calls that return a new `Query` object. As a first class object, this query can in turn be customized according to the needs of the application.

A query can be defined either *statically* or *dynamically*. A static query is defined in either annotation or XML metadata, and it must include both the query criteria as well as a user-assigned name. This kind of query is also called a *named query*, and it is later looked up by its name at the time it is executed.

A dynamic query can be issued at runtime by supplying only the JPQL query criteria. These may be a little more expensive to execute because the persistence provider cannot do any query preparation beforehand, but they are nevertheless very simple to use and can be issued in response to program logic or even user logic.

Following is an example showing how to create a query and then execute it to obtain all of the employees in the database. Of course this may not be a very good query to execute if the database is large and contains hundreds of thousands of employees, but it is nevertheless a legitimate example. The simple query is as follows:

```
Query query = em.createQuery("SELECT e FROM Employee e");
Collection emps = query.getResultList();
```

We create a `Query` object by issuing the `createQuery()` call on the `EntityManager` and passing in the JPQL string that specifies the query criteria. The JPQL string refers not to an `EMPLOYEE` database table but the `Employee` entity, so this query is selecting all `Employee` objects without filtering them any further. We will be diving into queries in Chapter 6 and JPQL in Chapters 6 and 7. You will see that you can be far more discretionary about which objects you want to be returned.

To execute the query we simply invoke `getResultList()` on it. This returns a `List` (a subinterface of `Collection`) containing the `Employee` objects that matched the query criteria. Note that a `List<Employee>` is not returned. Unfortunately this is not possible, since no class is passed into the call, so no parameterization of the type is able to occur. The return type is inferred by the persistence provider as it processes the JPQL string. We could cast the result to a `Collection<Employee>`, however, to make a neater return type for the caller. Doing so, we can easily create a method that returns all of the employees, as shown in Listing 2-8.

Listing 2-8. *Method for Issuing a Query*

```
public Collection<Employee> findAllEmployees() {
    Query query = em.createQuery("SELECT e FROM Employee e");
    return (Collection<Employee>) query.getResultList();
}
```

This example shows how simple queries are to create, execute, and process, but what this example does not show is how powerful they are. In Chapter 6 we will examine many other extremely useful and interesting ways of defining and using queries in an application.

Putting It All Together

We can now take all of the methods that we have created and combine them into a class. The class will act like a service class, which we will call `EmployeeService`, and will allow us to perform operations on employees. The code should be pretty familiar by now. Listing 2-9 shows the complete implementation.

Listing 2-9. *Service Class for Operating on* `Employee` *Entities*

```
import javax.persistence.*;
import java.util.Collection;
```

```
public class EmployeeService {
    protected EntityManager em;

    public EmployeeService(EntityManager em) {
        this.em = em;
    }

    public Employee createEmployee(int id, String name, long salary) {
        Employee emp = new Employee(id);
        emp.setName(name);
        emp.setSalary(salary);
        em.persist(emp);
        return emp;
    }

    public void removeEmployee(int id) {
        Employee emp = findEmployee(id);
        if (emp != null) {
            em.remove(emp);
        }
    }

    public Employee raiseEmployeeSalary(int id, long raise) {
        Employee emp = em.find(Employee.class, id);
        if (emp != null) {
            emp.setSalary(emp.getSalary() + raise);
        }
        return emp;
    }

    public Employee findEmployee(int id) {
        return em.find(Employee.class, id);
    }

    public Collection<Employee> findAllEmployees() {
        Query query = em.createQuery("SELECT e FROM Employee e");
        return (Collection<Employee>) query.getResultList();
    }
}
```

This is a simple yet fully functional class that can be used to issue the typical CRUD (create, read, update, and delete) operations on `Employee` entities. This class requires that an entity manager is created and passed into it by the caller and also that any required transactions are begun and committed by the caller. This may seem strange at first, but decoupling the transaction logic from the operation logic makes this class more portable to the Java EE environment. We will revisit this example in the next chapter, where we focus on Java EE applications.

A simple main program that uses this service and performs all of the required entity manager creation and transaction management is shown in Listing 2-10.

Listing 2-10. *Using* `EmployeeService`

```
import javax.persistence.*;
import java.util.Collection;

public class EmployeeTest {

    public static void main(String[] args) {
        EntityManagerFactory emf =
                Persistence.createEntityManagerFactory("EmployeeService");
        EntityManager em = emf.createEntityManager();
        EmployeeService service = new EmployeeService(em);

        //  create and persist an employee
        em.getTransaction().begin();
        Employee emp = service.createEmployee(158, "John Doe", 45000);
        em.getTransaction().commit();
        System.out.println("Persisted " + emp);

        // find a specific employee
        emp = service.findEmployee(158);
        System.out.println("Found " + emp);

        // find all employees
        Collection<Employee> emps = service.findAllEmployees();
        for (Employee e : emps)
            System.out.println("Found employee: " + e);

        // update the employee
        em.getTransaction().begin();
        emp = service.raiseEmployeeSalary(158, 1000);
        em.getTransaction().commit();
        System.out.println("Updated " + emp);

        // remove an employee
        em.getTransaction().begin();
        service.removeEmployee(158);
        em.getTransaction().commit();
        System.out.println("Removed Employee 158");

        // close the EM and EMF when done
        em.close();
        emf.close();
    }
}
```

Packaging It Up

Now that we know the basic building blocks of the Java Persistence API, we are ready to organize the pieces into an application that runs in Java SE. The only thing left to discuss is how to put it together so that it runs.

Persistence Unit

The configuration that describes the persistence unit is defined in an XML file called `persistence.xml`. Each persistence unit is named, so when a referencing application wants to specify the configuration for an entity it need only reference the name of the persistence unit that defines that configuration. A single `persistence.xml` file may contain one or more named persistence unit configurations, but each persistence unit is separate and distinct from the others, and they can be logically thought of as being in separate `persistence.xml` files.

Many of the persistence unit elements in the `persistence.xml` file apply to persistence units that are deployed within the Java EE container. The only ones that we need to specify for our example are `name`, `transaction-type`, `class`, and `properties`. There are a number of other elements that can be specified in the persistence unit configuration in the `persistence.xml` file, but these will be discussed in more detail in Chapter 11. Listing 2-11 shows the relevant part of the `persistence.xml` file for this example.

Listing 2-11. *Elements in the* `persistence.xml` *File*

```
<persistence>
    <persistence-unit name="EmployeeService" transaction-type="RESOURCE_LOCAL">
        <class>examples.model.Employee</class>
        <properties>
            <property name="toplink.jdbc.driver"
                      value="org.apache.derby.jdbc.ClientDriver"/>
            <property name="toplink.jdbc.url"
                      value="jdbc:derby://localhost:1527/EmpServDB;create=true"/>
            <property name="toplink.jdbc.user" value="APP"/>
            <property name="toplink.jdbc.password" value="APP"/>
        </properties>
    </persistence-unit>
</persistence>
```

The `name` element indicates the name of our persistence unit and is the string that we specify when we create the `EntityManagerFactory`. We have used "EmployeeService" as the name. The `transaction-type` element indicates that our persistence unit uses resource level `EntityTransaction` instead of JTA transactions. The `class` element lists the entity that is part of the persistence unit. Multiple `class` elements may be specified when there is more than one entity. These would not normally be needed when deploying in a Java EE container, but they are needed for portable execution when running in Java SE. We only have a single `Employee` entity.

The last part that we use is a list of properties that are vendor-specific. The login parameters to a database must be specified when running in a Java SE environment, so these properties exist to tell the provider what to connect to. Other provider properties, such as logging options, are also useful.

Persistence Archive

The persistence artifacts are packaged in what we will loosely call a *persistence archive*. This is really just a JAR-formatted file that contains the `persistence.xml` file in the `META-INF` directory and normally the entity class files.

Since we are running as a simple Java SE application, all we have to do is put the application JAR, the persistence provider JARs, and the Java Persistence API JAR on the classpath when the program is executed.

Summary

In this chapter we discussed just enough of the basics of the Java Persistence API to develop and run a simple application in a Java SE runtime.

We started out discussing the entity, how to define one, and how to turn an existing Java class into one. We discussed entity managers and how they are obtained and constructed in the Java SE environment.

The next step was to instantiate an entity instance and use the entity manager to persist it in the database. After we inserted some new entities, we were able to retrieve them again and then remove them. We also made some updates and ensured that the changes were written back to the database.

We talked about the resource-local transaction API and how to use it. We then went over some of the different types of queries and how to define and execute them. Finally, we aggregated all of these techniques and combined them into a simple application that we can execute in isolation from an enterprise environment.

In the next chapter, we will look at the impact of the Java EE environment when developing enterprise applications using the Java Persistence API.