Hibernate EntityManager

1

User guide

3.5.6-Final

by Emmanuel Bernard, Steve Ebersole, and Gavin King

Introducing JPA Persistence

The JPA specification recognizes the interest and the success of the transparent object/relational mapping paradigm. It standardizes the basic APIs and the metadata needed for any object/relational persistence mechanism. *Hibernate EntityManager* implements the programming interfaces and lifecycle rules as defined by the JPA 2.0 specification. Together with *Hibernate Annotations*, this wrapper implements a complete (and standalone) JPA persistence solution on top of the mature Hibernate Core. You may use a combination of all three together, annotations without JPA programming interfaces and lifecycle, or even pure native Hibernate Core, depending on the business and technical needs of your project. You can at all times fall back to Hibernate native APIs, or if required, even to native JDBC and SQL.

Architecture

1.1. Definitions

JPA 2 is part of the Java EE 6.0 platform. Persistence in JPA is available in containers like EJB 3 or the more modern CDI (Java Context and Dependency Injection), as well as in standalone Java SE applications that execute outside of a particular container. The following programming interfaces and artifacts are available in both environments.

`EntityManagerFactory`
: An entity manager factory provides entity manager instances, all instances are configured to connect to the same database, to use the same default settings as defined by the particular implementation, etc. You can prepare several entity manager factories to access several data stores. This interface is similar to the `SessionFactory` in native Hibernate.

`EntityManager`
: The `EntityManager` API is used to access a database in a particular unit of work. It is used to create and remove persistent entity instances, to find entities by their primary key identity, and to query over all entities. This interface is similar to the `Session` in Hibernate.

Persistence context
: A persistence context is a set of entity instances in which for any persistent entity identity there is a unique entity instance. Within the persistence context, the entity instances and their lifecycle is managed by a particular entity manager. The scope of this context can either be the transaction, or an extended unit of work.

Persistence unit
: The set of entity types that can be managed by a given entity manager is defined by a persistence unit. A persistence unit defines the set of all classes that are related or grouped by the application, and which must be collocated in their mapping to a single data store.

Container-managed entity manager
: An Entity Manager whose lifecycle is managed by the container

Application-managed entity manager
: An Entity Manager whose lifecycle is managed by the application.

JTA entity manager
: Entity manager involved in a JTA transaction

Resource-local entity manager
: Entity manager using a resource transaction (not a JTA transaction).

1.2. In container environment (eg. EJB 3)

1.2.1. Container-managed entity manager

The most common and widely used entity manager in a Java EE environment is the container-managed entity manager. In this mode, the container is responsible for the opening and closing of the entity manager (this is transparent to the application). It is also responsible for transaction boundaries. A container-managed entity manager is obtained in an application through dependency injection or through JNDI lookup, A container-managed entity manger requires the use of a JTA transaction.

1.2.2. Application-managed entity manager

An application-managed entity manager allows you to control the entity manager in application code. This entity manager is retrieved through the `EntityManagerFactory` API. An application managed entity manager can be either involved in the current JTA transaction (a JTA entity manager), or the transaction may be controlled through the `EntityTransaction` API (a resource-local entity manager). The resource-local entity manager transaction maps to a direct resource transaction (i. e. in Hibernate's case a JDBC transaction). The entity manager type (JTA or resource-local) is defined at configuration time, when setting up the entity manager factory.

1.2.3. Persistence context scope

An entity manager is the API to interact with the persistence context. Two common strategies can be used: binding the persistence context to the transaction boundaries, or keeping the persistence context available across several transactions.

The most common case is to bind the persistence context scope to the current transaction scope. This is only doable when JTA transactions are used: the persistence context is associated with the JTA transaction life cycle. When an entity manager is invoked, the persistence context is also opened, if there is no persistence context associated with the current JTA transaction. Otherwise, the associated persistence context is used. The persistence context ends when the JTA transaction completes. This means that during the JTA transaction, an application will be able to work on managed entities of the same persistence context. In other words, you don't have to pass the entity manager's persistence context across your managed beans (CDI) or EJBs method calls, but simply use dependency injection or lookup whenever you need an entity manager.

You can also use an extended persistence context. This can be combined with stateful session beans, if you use a container-managed entity manager: the persistence context is created when an entity manager is retrieved from dependency injection or JNDI lookup , and is kept until the container closes it after the completion of the `Remove` stateful session bean method. This is a perfect mechanism for implementing a "long" unit of work pattern. For example, if you have to deal with multiple user interaction cycles as a single unit of work (e.g. a wizard dialog that has to be fully completed), you usually model this as a unit of work from the point of view of the application user, and implement it using an extended persistence context. Please refer to the Hibernate reference manual or the book Hibernate In Action for more information about this pattern.

JBoss Seam 3 is built on top of CDI and has at it's core concept the notion of conversation and unit of work. For an application-managed entity manager the persistence context is created when the entity manager is created and kept until the entity manager is closed. In an extended persistence context, all modification operations (persist, merge, remove) executed outside a transaction are queued until the persistence context is attached to a transaction. The transaction typically occurs at the user process end, allowing the whole process to be committed or rollbacked. For application-managed entity manager only support the extended persistence context.

A resource-local entity manager or an entity manager created with `EntityManagerFactory.createEntityManager()` (application-managed) has a one-to-one relationship with a persistence context. In other situations *persistence context propagation* occurs.

1.2.4. Persistence context propagation

Persistence context propagation occurs for container-managed entity managers.

In a transaction-scoped container managed entity manager (common case in a Java EE environment), the JTA transaction propagation is the same as the persistence context resource propagation. In other words, container-managed transaction-scoped entity managers retrieved within a given JTA transaction all share the same persistence context. In Hibernate terms, this means all managers share the same session.

Important: persistence context are never shared between different JTA transactions or between entity manager that do not came from the same entity manager factory. There are some noteworthy exceptions for context propagation when using extended persistence contexts:

- If a stateless session bean, message-driven bean, or stateful session bean with a transaction-scoped persistence context calls a stateful session bean with an extended persistence context in the same JTA transaction, an `IllegalStateException` is thrown.
- If a stateful session bean with an extended persistence context calls as stateless session bean or a stateful session bean with a transaction-scoped persistence context in the same JTA transaction, the persistence context is propagated.
- If a stateful session bean with an extended persistence context calls a stateless or stateful session bean in a different JTA transaction context, the persistence context is not propagated.
- If a stateful session bean with an extended persistence context instantiates another stateful session bean with an extended persistence context, the extended persistence context is inherited by the second stateful session bean. If the second stateful session bean is called with a different transaction context than the first, an IllegalStateException is thrown.
- If a stateful session bean with an extended persistence context calls a stateful session bean with a different extended persistence context in the same transaction, an `IllegalStateException` is thrown.

1.3. Java SE environments

In a Java SE environment only extended context application-managed entity managers are available. You can retrieve an entity manger using the `EntityManagerFactory` API. Only resource-local entity managers are available. In other words, JTA transactions and persistence context propagation are not supported in Java SE (you will have to propagate the persistence context yourself, e.g. using the thread local session pattern popular in the Hibernate community).

Extended context means that a persistence context is created when the entity manager is retrieved (using `EntityManagerFactory.createEntityManager(...)`) and closed when the entity manager is closed. Many resource-local transaction share the same persistence context, in this case.

Setup and configuration

2.1. Setup

The JPA 2.0 compatible Hibernate EntityManager is built on top of the core of Hibernate and Hibernate Annotations. Starting from version 3.5, we have bundled in a single Hibernate distribution all the necessary modules:

- Hibernate Core: the native Hibernate APIs and core engine
- Hibernate Annotations: the annotation-based mapping
- Hibernate EntityManager: the JPA 2.0 APIs and livecycle semantic implementation

Download the Hibernate Core distribution. Set up your classpath (after you have created a new project in your favorite IDE):

- Copy `hibernate3.jar` and the required 3rd party libraries available in `lib/required`.
- Copy `lib/jpa/hibernate-jpa-2.0-api-1.0.0.Final.jar` to your classpath as well.

What is hibernate-jpa-2.0-api-x.y.z.jar?

This is the JAR containing the JPA 2.0 API, it provides all the interfaces and concrete classes that the specification defines as public API. Said otherwise, you can use this JAR to bootstrap any JPA provider implementation. Note that you typically don't need it when you deploy your application in a Java EE 6 application server (like JBoss AS 6 for example).

Alternatively, if you use Maven, add the following dependencies

```
<project ...>
  ...
  <dependencies>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-entitymanager</artifactId>
      <version>${hibernate-core-version}</version>
    </dependency>
  </dependencies>
</project>
```

All the required dependencies like hibernate-core and hibernate-annotations will be dragged transitively.

We recommend you use *Hibernate Validator* [http://validator.hibernate.org] and the Bean Validation specification capabilities as its integration with Java Persistence 2 has been

standardized. Download Hibernate Validator 4 or above from the Hibernate website and add `hibernate-validator.jar` and `validation-api.jar` in your classpath. Alternatively add the following dependency in your `pom.xml`.

```
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-validator</artifactId>
      <version>${hibernate-validator-version}</version>
    </dependency>
    ...
  </dependencies>
  ...
</project>
```

If you wish to use *Hibernate Search* [http://search.hibernate.org] (full-text search for Hibernate aplications), download it from the Hibernate website and add `hibernate-search.jar` and its dependencies in your classpath. Alternatively add the following dependency in your `pom.xml`.

```
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-search</artifactId>
      <version>${hibernate-search-version}</version>
    </dependency>
    ...
  </dependencies>
  ...
</project>
```

2.2. Configuration and bootstrapping

2.2.1. Packaging

The configuration for entity managers both inside an application server and in a standalone application reside in a persistence archive. A persistence archive is a JAR file which must define a `persistence.xml` file that resides in the `META-INF` folder. All properly annotated classes included in the archive (ie. having an `@Entity` annotation), all annotated packages and all Hibernate hbm.xml files included in the archive will be added to the persistence unit configuration, so by default, your persistence.xml will be quite minimalist:

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

```
               xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/
xml/ns/persistence/persistence_2_0.xsd"
            version="2.0">
   <persistence-unit name="sample">
      <jta-data-source>java:/DefaultDS</jta-data-source>
      <properties>
         <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect"/>
         <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
      </properties>
   </persistence-unit>
</persistence>
```

Here's a more complete example of a `persistence.xml` file

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/
xml/ns/persistence/persistence_2_0.xsd"
             version="2.0">
   <persistence-unit name="manager1" transaction-type="JTA">
      <provider>org.hibernate.ejb.HibernatePersistence</provider>
      <jta-data-source>java:/DefaultDS</jta-data-source>
      <mapping-file>ormap.xml</mapping-file>
      <jar-file>MyApp.jar</jar-file>
      <class>org.acme.Employee</class>
      <class>org.acme.Person</class>
      <class>org.acme.Address</class>
      <shared-cache-mode>ENABLE_SELECTOVE</shared-cache-mode>
      <validation-mode>CALLBACK</validation-mode>
      <properties>
         <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect"/>
         <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
      </properties>
   </persistence-unit>
</persistence>
```

`name`
: (attribute) Every entity manager must have a name.

`transaction-type`
: (attribute) Transaction type used. Either JTA or RESOURCE_LOCAL (default to JTA in a JavaEE environment and to RESOURCE_LOCAL in a JavaSE environment). When a jta-datasource is used, the default is JTA, if non-jta-datasource is used, RESOURCE_LOCAL is used.

`provider`
: The provider is a fully-qualified class name of the EJB Persistence provider. You do not have to define it if you don't work with several EJB3 implementations. This is needed when you are using multiple vendor implementations of EJB Persistence.

`jta-data-source`, `non-jta-data-source`
: This is the JNDI name of where the javax.sql.DataSource is located. When running without a JNDI available Datasource, you must specify JDBC connections with Hibernate specific properties (see below).

`mapping-file`
: The class element specifies a EJB3 compliant XML mapping file that you will map. The file has to be in the classpath. As per the EJB3 specification, Hibernate EntityManager will try to load the mapping file located in the jar file at `META_INF/orm.xml`. Of course any explicit mapping file will be loaded too. As a matter of fact, you can provides any XML file in the mapping file element ie. either hbm files or EJB3 deployment descriptor.

`jar-file`
: The jar-file elements specifies a jar to analyse. All properly annotated classes, annotated packages and all hbm.xml files part of this jar file will be added to the persistence unit configuration. This element is mainly used in Java EE environment. Use of this one in Java SE should be considered as non portable, in this case a absolute url is needed. You can alternatively point to a directory (This is especially useful when in your test environment, the persistence.xml file is not under the same root directory or jar than your domain model).

```
        <jar-file>file:/home/turin/work/local/lab8/build/classes</jar-file>
```

`exclude-unlisted-classes`
: Do not check the main jar file for annotated classes. Only explicit classes will be part of the persistence unit.

`class`
: The class element specifies a fully qualified class name that you will map. By default all properly annotated classes and all hbm.xml files found inside the archive are added to the persistence unit configuration. You can add some external entity through the class element though. As an extension to the specification, you can add a package name in the `<class>` element (eg `<class>org.hibernate.eg</class>`). Caution, the package will include the metadata defined at the package level (ie in `package-info.java`), it will not include all the classes of a given package.

shared-cache-mode
: By default, entities are elected for second-level cache if annotated with `@Cacheable`. You can however:

  - `ALL`: force caching for all entities
  - `NONE`: disable caching for all entities (useful to take second-level cache out of the equation)
  - `ENABLE_SELECTIVE` (default): enable caching when explicitly marked
  - `DISABLE_SELECTIVE`: enable caching unless explicitly marked as `@Cacheable(false)` (not recommended)

See Hibernate Annotation's documentation for more details.

validation-mode

By default, Bean Validation (and Hibernate Validator) is activated. When an entity is created, updated (and optionally deleted), it is validated before being sent to the database. The database schema generated by Hibernate also reflects the constraints declared on the entity.

You can fine-tune that if needed:

- `AUTO`: if Bean Validation is present in the classpath, CALLBACK and DDL are activated.
- `CALLBACK`: entities are validated on creation, update and deletion. If no Bean Validation provider is present, an exception is raised at initialization time.
- `DDL`: (not standard, see below) database schemas are entities are validated on creation, update and deletion. If no Bean Validation provider is present, an exception is raised at initialization time.
- `NONE`: Bean Validation is not used at all

Unfortunately, `DDL` is not standard mode (though extremely useful) and you will not be able to put it in `<validation-mode>`. To use it, add a regular property

```
<property name="javax.persistence.validation.mode">
  ddl
</property>
```

With this approach, you can mix ddl and callback modes:

```
<property name="javax.persistence.validation.mode">
  ddl, callback
</property>
```

`properties`

The properties element is used to specify vendor specific properties. This is where you will define your Hibernate specific configurations. This is also where you will have to specify JDBC connection information as well.

Here is a list of JPA 2 standard properties. Be sure to also Hibernate Core's documentation to see Hibernate specific properties.

- `javax.persistence.lock.timeout` pessimistic lock timeout in milliseconds (`Integer` or `String`), this is a hint used by Hibernate but requires support by your underlying database.
- `javax.persistence.query.timeout` query timeout in milliseconds (`Integer` or `String`), this is a hint used by Hibernate but requires support by your underlying database (TODO is that 100% true or do we use some other tricks).

- `javax.persistence.validation.mode` corresponds to the `validation-mode` element. Use it if you wish to use the non standard `DDL` value.
- `javax.persistence.validation.group.pre-persist` defines the group or list of groups to validate before persisting an entity. This is a comma separated fully qualified class name string (eg `com.acme.groups.Common` or `com.acme.groups.Common, javax.validation.groups.Default`). Defaults to the Bean Validation default group.
- `javax.persistence.validation.group.pre-update` defines the group or list of groups to validate before updating an entity. This is a comma separated fully qualified class name string (eg `com.acme.groups.Common` or `com.acme.groups.Common, javax.validation.groups.Default`). Defaults to the Bean Validation default group.
- `javax.persistence.validation.group.pre-remove` defines the group or list of groups to validate before persisting an entity. This is a comma separated fully qualified class name string (eg `com.acme.groups.Common` or `com.acme.groups.Common, javax.validation.groups.Default`). Defaults to no group.

Note

To know more about Bean Validation and Hibernate Validator, check out Hibernate Validator's reference documentation as well as Hibernate Annotations's documentation on Bean Validation.

The following properties can only be used in a SE environment where no datasource/JNDI is available:

- `javax.persistence.jdbc.driver`: the fully qualified class name of the driver class
- `javax.persistence.jdbc.url`: the driver specific URL
- `javax.persistence.jdbc.user` the user name used for the database connection
- `javax.persistence.jdbc.password` the password used for the database connection

Be sure to define the grammar definition in the `persistence` element since the JPA specification requires schema validation. If the `systemId` ends with `persistence_2_0.xsd`, Hibernate entityManager will use the version embedded in the hibernate-entitymanager.jar. It won't fetch the resource from the internet.

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/
xml/ns/persistence/persistence_2_0.xsd"
             version="2.0">
```

2.2.2. Bootstrapping

The JPA specification defines a bootstrap procedure to access the `EntityManagerFactory` and the `EntityManager`. The bootstrap class is `javax.persistence.Persistence`, e.g.

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("manager1");

//or

Map<String, Object> configOverrides = new HashMap<String, Object>();
configOverrides.put("hibernate.hbm2ddl.auto", "create-drop");
EntityManagerFactory programmaticEmf =
    Persistence.createEntityManagerFactory("manager1", configOverrides);
```

The first version is equivalent to the second with an empty map. The map version is a set of overrides that will take precedence over any properties defined in your `persistence.xml` files. All the properties defined in *Section 2.2.1, "Packaging"* can be passed to the `createEntityManagerFactory` method and there are a few additional ones:

- `javax.persistence.provider` to define the provider class used
- `javax.persistence.transactionType` to define the transaction type used (either `JTA` or `RESOURCE_LOCAL`)
- `javax.persistence.jtaDataSource` to define the JTA datasource name in JNDI
- `javax.persistence.nonJtaDataSource` to define the non JTA datasource name in JNDI
- `javax.persistence.lock.timeout` pessimistic lock timeout in milliseconds (`Integer` or `String`)
- `javax.persistence.query.timeout` query timeout in milliseconds (`Integer` or `String`)
- `javax.persistence.sharedCache.mode` corresponds to the `share-cache-mode` element defined in *Section 2.2.1, "Packaging"*.
- `javax.persistence.validation.mode` corresponds to the `validation-mode` element defined in *Section 2.2.1, "Packaging"*.

When `Persistence.createEntityManagerFactory()` is called, the persistence implementation will search your classpath for any `META-INF/persistence.xml` files using the `ClassLoader.getResource("META-INF/persistence.xml")` method. Actually the `Persistence` class will look at all the Persistence Providers available in the classpath and ask each of them if they are responsible for the creation of the entity manager factory `manager1`. Each provider, from this list of resources, it will try to find an entity manager that matches the name you specify in the command line with what is specified in the persistence.xml file (of course the provider `element` must match the current persistent provider). If no persistence.xml with the correct name are found or if the expected persistence provider is not found, a `PersistenceException` is raised.

Apart from Hibernate system-level settings, all the properties available in Hibernate can be set in `properties` element of the persistence.xml file or as an override in the map you pass to `createEntityManagerFactory()`. Please refer to the Hibernate reference documentation for a complete listing. There are however a couple of properties available in the EJB3 provider only.

Table 2.1. Hibernate Entity Manager specific properties

Property name	Description
hibernate.ejb.classcache.<classname>	class cache strategy [comma cache region] of the class Default to no cache, and default region cache to fully.qualified.classname (eg. hibernate.ejb.classcache.com.acme.Cat read-write or hibernate.ejb.classcache.com.acme.Cat read-write, MyRegion).
hibernate.ejb.collectioncache.<collectionrole>	collection cache strategy [comma cache region] of the class Default to no cache, and default region cache to fully.qualified.classname.role (eg. hibernate.ejb.classcache.com.acme.Cat read-write or hibernate.ejb.classcache.com.acme.Cat read-write, MyRegion).
hibernate.ejb.cfgfile	XML configuration file to use to configure Hibernate (eg. `/hibernate.cfg.xml`).
hibernate.archive.autodetection	Determine which element is auto discovered by Hibernate Entity Manager while parsing the .par archive. (default to `class,hbm`).
hibernate.ejb.interceptor	An optional Hibernate interceptor. The interceptor instance is shared by all `Session` instances. This interceptor has to implement `org.hibernate.Interceptor` and have a no-arg constructor. This property can not be combined with `hibernate.ejb.interceptor.session_scoped`.
hibernate.ejb.interceptor.session_scoped	An optional Hibernate interceptor. The interceptor instance is specific to a given `Session` instance (and hence can be non thread-safe). This interceptor has to implement `org.hibernate.Interceptor` and have a no-arg constructor. This property can not be combined with `hibernate.ejb.interceptor`.
hibernate.ejb.naming_strategy	An optional naming strategy. The default naming strategy used is `EJB3NamingStrategy`. You also might want to consider the `DefaultComponentSafeNamingStrategy`.
hibernate.ejb.event.<eventtype>	Event listener list for a given eventtype. The list of event listeners is a comma separated fully qualified class name list (eg. hibernate.ejb.event.pre-load com.acme.SecurityListener, com.acme.AuditListener)

Property name	Description
hibernate.ejb.use_class_enhancer	Whether or not use Application server class enhancement at deployment time (default to false)
hibernate.ejb.discard_pc_on_close	If true, the persistence context will be discarded (think clear() when the method is called. Otherwise the persistence context will stay alive till the transaction completion: all objects will remain managed, and any change will be synchronized with the database (default to false, ie wait the transaction completion)
hibernate.ejb.resource_scanner	By default, Hibernate EntityManager scans itself the list of resources for annotated classes and persistence deployment descriptors (like orm.xml and hbm.xml files). You can customize this scanning strategy by implementing `org.hibernate.ejb.packaging.Scanner`. This property is used by container implementors to improve integration with Hibernate. Accepts an instance of `Scanner` or the file name of a no-arg constructor class implementing `Scanner`.

Note that you can mix XML `<class>` declaration and `hibernate.ejb.cfgfile` usage in the same configuration. Be aware of the potential clashed. The properties set in `persistence.xml` will override the one in the defined `hibernate.cfg.xml`.

Note

It is important that you do not override `hibernate.transaction.factory_class`, Hibernate EntityManager automatically set the appropriate transaction factory depending on the EntityManager type (ie `JTA` versus `RESOURSE_LOCAL`). If you are working in a Java EE environment, you might want to set the `hibernate.transaction.manager_lookup_class` though.

Here is a typical configuration in a Java SE environment

```
<persistence>
   <persistence-unit name="manager1" transaction-type="RESOURCE_LOCAL">
      <class>org.hibernate.ejb.test.Cat</class>
      <class>org.hibernate.ejb.test.Distributor</class>
      <class>org.hibernate.ejb.test.Item</class>
      <properties>
         <property name="javax.persistence.jdbc.driver" value="org.hsqldb.jdbcDriver"/>
         <property name="javax.persistence.jdbc.user" value="sa"/>
         <property name="javax.persistence.jdbc.password" value=""/>
         <property name="javax.persistence.jdbc.url" value="jdbc:hsqldb:."/>
         <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect"/
         <property name="hibernate.max_fetch_depth" value="3"/>
```

```
         <!-- cache configuration -->
            <property name="hibernate.ejb.classcache.org.hibernate.ejb.test.Item" value="read-
write"/>

<property name="hibernate.ejb.collectioncache.org.hibernate.ejb.test.Item.distributors" value="read-
write, RegionName"/>

            <!-- alternatively to <class> and <property> declarations, you can use a regular
 hibernate.cfg.xml file -->
                 <!--  property  name="hibernate.ejb.cfgfile"  value="/org/hibernate/ejb/test/
hibernate.cfg.xml"/ -->
      </properties>
   </persistence-unit>
</persistence>
```

To ease the programmatic configuration, Hibernate Entity Manager provide a proprietary API. This API is very similar to the `Configuration` API and share the same concepts: `Ejb3Configuration`. Refer to the JavaDoc and the Hibernate reference guide for more detailed informations on how to use it.

TODO: me more descriptive on some APIs like setDatasource()

```
Ejb3Configuration cfg = new Ejb3Configuration();
EntityManagerFactory emf =
  cfg.addProperties( properties ) //add some properties
     .setInterceptor( myInterceptorImpl ) // set an interceptor
     .addAnnotatedClass( MyAnnotatedClass.class ) //add a class to be mapped
     .addClass( NonAnnotatedClass.class ) //add an hbm.xml file using the Hibernate convention
     .addRerousce( "mypath/MyOtherCLass.hbm.xml ) //add an hbm.xml file
     .addRerousce( "mypath/orm.xml ) //add an EJB3 deployment descriptor
     .configure("/mypath/hibernate.cfg.xml") //add a regular hibernate.cfg.xml
     .buildEntityManagerFactory(); //Create the entity manager factory
```

2.3. Event listeners

Hibernate Entity Manager needs to enhance Hibernate core to implements all the JPA semantics. It does that through the event listener system of Hibernate. Be careful when you use the event system yourself, you might override some of the JPA semantics. A safe way is to add your event listeners to the list given below.

Table 2.2. Hibernate Entity Manager default event listeners

Event	Listeners
flush	org.hibernate.ejb.event.EJB3FlushEventListener
auto-flush	org.hibernate.ejb.event.EJB3AutoFlushEventListener
delete	org.hibernate.ejb.event.EJB3DeleteEventListener
flush-entity	org.hibernate.ejb.event.EJB3FlushEntityEventListener

Event	Listeners
merge	org.hibernate.ejb.event.EJB3MergeEventListener
create	org.hibernate.ejb.event.EJB3PersistEventListener
create-onflush	org.hibernate.ejb.event.EJB3PersistOnFlushEventListener
save	org.hibernate.ejb.event.EJB3SaveEventListener
save-update	org.hibernate.ejb.event.EJB3SaveOrUpdateEventListener
pre-insert	org.hibernate.secure.JACCPreInsertEventListener
pre-insert	org.hibernate.secure.JACCPreUpdateEventListener
pre-delete	org.hibernate.secure.JACCPreDeleteEventListener
pre-load	org.hibernate.secure.JACCPreLoadEventListener
post-delete	org.hibernate.ejb.event.EJB3PostDeleteEventListener
post-insert	org.hibernate.ejb.event.EJB3PostInsertEventListener
post-load	org.hibernate.ejb.event.EJB3PostLoadEventListener
post-update	org.hibernate.ejb.event.EJB3PostUpdateEventListener

Note that the `JACC*EventListeners` are removed if the security is not enabled.

You can configure the event listeners either through the properties (see *Configuration and bootstrapping*) or through the `ejb3configuration.getEventListeners()` API.

2.4. Obtaining an EntityManager in a Java SE environment

An entity manager factory should be considered as an immutable configuration holder, it is defined to point to a single datasource and to map a defined set of entities. This is the entry point to create and manage `EntityManagers`. The `Persistence` class is bootstrap class to create an entity manager factory.

```
// Use persistence.xml configuration
EntityManagerFactory emf = Persistence.createEntityManagerFactory("manager1")
EntityManager em = emf.createEntityManager(); // Retrieve an application managed entity manager
// Work with the EM
em.close();
...
emf.close(); //close at application end
```

An entity manager factory is typically create at application initialization time and closed at application end. It's creation is an expensive process. For those who are familiar with Hibernate, an entity manager factory is very much like a session factory. Actually, an entity manager factory is a wrapper on top of a session factory. Calls to the entityManagerFactory are thread safe.

Thanks to the `EntityManagerFactory`, you can retrieve an extended entity manager. The extended entity manager keep the same persistence context for the lifetime of the entity manager: in other words, the entities are still managed between two transactions (unless you call `entityManager.clear()` in between). You can see an entity manager as a small wrapper on top of an Hibernate session.

TODO explains emf.createEntityManager(Map)

2.5. Various

Hibernate Entity Manager comes with Hibernate Validator configured out of the box. You don't have to override any event yourself. If you do not use Hibernate Validator annotations in your domain model, there will be no performance cost. For more information on Hibernate Validator, please refer to the Hibernate Annotations reference guide.

Working with objects

3.1. Entity states

Like in Hibernate (comparable terms in parentheses), an entity instance is in one of the following states:

- New (transient): an entity is new if it has just been instantiated using the new operator, and it is not associated with a persistence context. It has no persistent representation in the database and no identifier value has been assigned.
- Managed (persistent): a managed entity instance is an instance with a persistent identity that is currently associated with a persistence context.
- Detached: the entity instance is an instance with a persistent identity that is no longer associated with a persistence context, usually because the persistence context was closed or the instance was evicted from the context.
- Removed: a removed entity instance is an instance with a persistent identity, associated with a persistence context, but scheduled for removal from the database.

The `EntityManager` API allows you to change the state of an entity, or in other words, to load and store objects. You will find persistence with JPA easier to understand if you think about object state management, not managing of SQL statements.

3.2. Making objects persistent

Once you've created a new entity instance (using the common `new` operator) it is in `new` state. You can make it persistent by associating it to an entity manager:

```
DomesticCat fritz = new DomesticCat();
fritz.setColor(Color.GINGER);
fritz.setSex('M');
fritz.setName("Fritz");
em.persist(fritz);
```

If the `DomesticCat` entity type has a generated identifier, the value is associated to the instance when `persist()` is called. If the identifier is not automatically generated, the application-assigned (usually natural) key value has to be set on the instance before `persist()` is called.

3.3. Loading an object

Load an entity instance by its identifier value with the entity manager's `find()` method:

```
cat = em.find(Cat.class, catId);
```

```
// You may need to wrap the primitive identifiers
long catId = 1234;
em.find( Cat.class, new Long(catId) );
```

In some cases, you don't really want to load the object state, but just having a reference to it (ie a proxy). You can get this reference using the `getReference()` method. This is especially useful to link a child to its parent without having to load the parent.

```
child = new Child();
child.SetName("Henry");
Parent parent = em.getReference(Parent.class, parentId); //no query to the DB
child.setParent(parent);
em.persist(child);
```

You can reload an entity instance and it's collections at any time using the `em.refresh()` operation. This is useful when database triggers are used to initialize some of the properties of the entity. Note that only the entity instance and its collections are refreshed unless you specify `REFRESH` as a cascade style of any associations:

```
em.persist(cat);
em.flush(); // force the SQL insert and triggers to run
em.refresh(cat); //re-read the state (after the trigger executes)
```

3.4. Querying objects

If you don't know the identifier values of the objects you are looking for, you need a query. The Hibernate EntityManager implementation supports an easy-to-use but powerful object-oriented query language (JP-QL) which has been inspired by HQL (and vice-versa). HQL is strictly speaking a superset of JP-QL. Both query languages are portable across databases, the use entity and property names as identifiers (instead of table and column names). You may also express your query in the native SQL of your database, with optional support from JPA for result set conversion into Java business objects.

3.4.1. Executing queries

JP-QL and SQL queries are represented by an instance of `javax.persistence.Query`. This interface offers methods for parameter binding, result set handling, and for execution of the query. Queries are always created using the current entity manager:

```
List<?> cats = em.createQuery(
    "select cat from Cat as cat where cat.birthdate < ?1")
    .setParameter(1, date, TemporalType.DATE)
    .getResultList();

List<?> mothers = em.createQuery(
```

```
    "select mother from Cat as cat join cat.mother as mother where cat.name = ?1")
    .setParameter(1, name)
    .getResultList();

List<?> kittens = em.createQuery(
    "from Cat as cat where cat.mother = ?1")
    .setEntity(1, pk)
    .getResultList();

Cat mother = (Cat) em.createQuery(
    "select cat.mother from Cat as cat where cat = ?1")
    .setParameter(1, izi)
    .getSingleResult();
```

A query is usually executed by invoking `getResultList()`. This method loads the resulting instances of the query completely into memory. Entity instances retrieved by a query are in persistent state. The `getSingleResult()` method offers a shortcut if you know your query will only return a single object.

JPA 2 provides more type-safe approaches to queries. The truly type-safe approach is the Criteria API explained in *Chapter 9, Criteria Queries*.

```
CriteriaQuery<Cat> criteria = builder.createQuery( Cat.class );
Root<Cat> cat = criteria.from( Cat.class );
criteria.select( cat );
criteria.where( builder.lt( cat.get( Cat_.birthdate ), catDate ) );
List<Cat> cats = em.createQuery( criteria ).getResultList(); //notice no downcasting is necessary
```

But you can benefit form some type-safe convenience even when using JP-QL (note that it's not as type-safe as the compiler has to trust you with the return type.

```
//No downcasting since we pass the return type
List<Cat> cats = em.createQuery(
    "select cat from Cat as cat where cat.birthdate < ?1", Cat.class)
    .setParameter(1, date, TemporalType.DATE)
    .getResultList();
```

Note

We highly recommend the Criteria API approach. While more verbose, it provides compiler-enforced safety (including down to property names) which will pay off when the application will move to maintenance mode.

3.4.1.1. Projection

JPA queries can return tuples of objects if projection is used. Each result tuple is returned as an object array:

```
Iterator kittensAndMothers = sess.createQuery(
            "select kitten, mother from Cat kitten join kitten.mother mother")
            .getResultList()
            .iterator();

while ( kittensAndMothers.hasNext() ) {
    Object[] tuple = (Object[]) kittensAndMothers.next();
    Cat kitten = (Cat) tuple[0];
    Cat mother = (Cat) tuple[1];
    ....
}
```

Note

The criteria API provides a type-safe approach to projection results. Check out *Section 9.2, "Tuple criteria queries"*.

3.4.1.2. Scalar results

Queries may specify a particular property of an entity in the select clause, instead of an entity alias. You may call SQL aggregate functions as well. Returned non-transactional objects or aggregation results are considered "scalar" results and are not entities in persistent state (in other words, they are considered "read only"):

```
Iterator results = em.createQuery(
        "select cat.color, min(cat.birthdate), count(cat) from Cat cat " +
        "group by cat.color")
        .getResultList()
        .iterator();

while ( results.hasNext() ) {
    Object[] row = results.next();
    Color type = (Color) row[0];
    Date oldest = (Date) row[1];
    Integer count = (Integer) row[2];
    .....
}
```

3.4.1.3. Bind parameters

Both named and positional query parameters are supported, the `Query` API offers several methods to bind arguments. The JPA specification numbers positional parameters from one. Named

parameters are identifiers of the form `:paramname` in the query string. Named parameters should be preferred, they are more robust and easier to read and understand:

```
// Named parameter (preferred)
Query q = em.createQuery("select cat from DomesticCat cat where cat.name = :name");
q.setParameter("name", "Fritz");
List cats = q.getResultList();

// Positional parameter
Query q = em.createQuery("select cat from DomesticCat cat where cat.name = ?1");
q.setParameter(1, "Izi");
List cats = q.getResultList();

// Named parameter list
List names = new ArrayList();
names.add("Izi");
names.add("Fritz");
Query q = em.createQuery("select cat from DomesticCat cat where cat.name in (:namesList)");
q.setParameter("namesList", names);
List cats = q.list();
```

3.4.1.4. Pagination

If you need to specify bounds upon your result set (the maximum number of rows you want to retrieve and/or the first row you want to retrieve), use the following methods:

```
Query q = em.createQuery("select cat from DomesticCat cat");
q.setFirstResult(20);
q.setMaxResults(10);
List cats = q.getResultList(); //return cats from the 20th position to 29th
```

Hibernate knows how to translate this limit query into the native SQL of your DBMS.

3.4.1.5. Externalizing named queries

You may also define named queries through annotations:

```
@javax.persistence.NamedQuery(name="eg.DomesticCat.by.name.and.minimum.weight",
  query="select cat from eg.DomesticCat as cat  where cat.name = ?1 and cat.weight > ?2")
```

Parameters are bound programmatically to the named query, before it is executed:

```
Query q = em.createNamedQuery("eg.DomesticCat.by.name.and.minimum.weight");
q.setString(1, name);
q.setInt(2, minWeight);
List<?> cats = q.getResultList();
```

You can also use the slightly more type-safe approach:

```
Query q = em.createNamedQuery("eg.DomesticCat.by.name.and.minimum.weight", Cat.class);
q.setString(1, name);
q.setInt(2, minWeight);
List<Cat> cats = q.getResultList();
```

Note that the actual program code is independent of the query language that is used, you may also define native SQL queries in metadata, or use Hibernate's native facilities by placing them in XML mapping files.

3.4.1.6. Native queries

You may express a query in SQL, using `createNativeQuery()` and let Hibernate take care mapping from JDBC result sets to business objects. Use the `@SqlResultSetMapping` (please see the Hibernate Annotations reference documentation on how to map a SQL resultset mapping) or the entity mapping (if the column names of the query result are the same as the names declared in the entity mapping; remember that all entity columns have to be returned for this mechanism to work):

```
@SqlResultSetMapping(name="getItem", entities =
        @EntityResult(entityClass=org.hibernate.ejb.test.Item.class, fields= {
            @FieldResult(name="name", column="itemname"),
            @FieldResult(name="descr", column="itemdescription")
        })
)

Query q = em.createNativeQuery("select name as itemname, descr as itemdescription from
 Item", "getItem");
item = (Item) q.getSingleResult(); //from a resultset

Query q = em.createNativeQuery("select * from Item", Item.class);
item = (Item) q.getSingleResult(); //from a class columns names match the mapping
```

Note

For more information about scalar support in named queries, please refers to the Hibernate Annotations documentation

3.4.1.7. Query lock and flush mode

You can adjust the flush mode used when executing the query as well as define the lock mode used to load the entities.

Adjusting the flush mode is interesting when one must guaranty that a query execution will not trigger a flush operation. Most of the time you don't need to care about this.

Adjusting the lock mode is useful if you need to lock the objects returns by the query to a certain level.

```
query.setFlushMode(FlushModeType.COMMIT)
     .setLockMode(LockModeType.PESSIMISTIC_READ);
```

Note

If you want to use `FlushMode.MANUAL` (ie the Hibernate specific flush mode), you will need to use a query hint. See below.

3.4.1.8. Query hints

Query hints (for performance optimization, usually) are implementation specific. Hints are declared using the `query.setHint(String name, Object value)` method, or through the `@Named(Native)Query(hints)` annotation Note that these are not SQL query hints! The Hibernate EJB3 implementation offers the following query hints:

Table 3.1. Hibernate query hints

Hint	Description
org.hibernate.timeout	Query timeout in seconds (eg. new Integer(10))
org.hibernate.fetchSize	Number of rows fetched by the JDBC driver per roundtrip (eg. new Integer(50))
org.hibernate.comment	Add a comment to the SQL query, useful for the DBA (e.g. new String("fetch all orders in 1 statement"))
org.hibernate.cacheable	Whether or not a query is cacheable (eg. new Boolean(true)), defaults to false
org.hibernate.cacheMode	Override the cache mode for this query (eg. CacheMode.REFRESH)
org.hibernate.cacheRegion	Cache region of this query (eg. new String("regionName"))
org.hibernate.readOnly	Entities retrieved by this query will be loaded in a read-only mode where Hibernate will never dirty-check them or make changes persistent (eg. new Boolean(true)), default to false
org.hibernate.flushMode	Flush mode used for this query (useful to pass Hibernate specific flush modes, in particular `MANUAL`).

Hint	Description
org.hibernate.cacheMode	Cache mode used for this query

The value object accept both the native type or its string equivalent (eg. `CaheMode.REFRESH` or "`REFRESH`"). Please refer to the Hibernate reference documentation for more information.

3.5. Modifying persistent objects

Transactional managed instances (ie. objects loaded, saved, created or queried by the entity manager) may be manipulated by the application and any changes to persistent state will be persisted when the Entity manager is flushed (discussed later in this chapter). There is no need to call a particular method to make your modifications persistent. A straightforward wayt to update the state of an entity instance is to `find()` it, and then manipulate it directly, while the persistence context is open:

```
Cat cat = em.find( Cat.class, new Long(69) );
cat.setName("PK");
em.flush();  // changes to cat are automatically detected and persisted
```

Sometimes this programming model is inefficient since it would require both an SQL SELECT (to load an object) and an SQL UPDATE (to persist its updated state) in the same session. Therefore Hibernate offers an alternate approach, using detached instances.

3.6. Detaching a object

An object when loaded in the persistence context is managed by Hibernate. You can force an object to be detached (ie. no longer managed by Hibernate) by closing the EntityManager or in a more fine-grained approach by calling the `detach()` method.

```
Cat cat = em.find( Cat.class, new Long(69) );
...
em.detach(cat);
cat.setName("New name"); //not propatated to the database
```

3.7. Modifying detached objects

Many applications need to retrieve an object in one transaction, send it to the presentation layer for manipulation, and later save the changes in a new transaction. There can be significant user think and waiting time between both transactions. Applications that use this kind of approach in a high-concurrency environment usually use versioned data to ensure isolation for the "long" unit of work.

The JPA specifications supports this development model by providing for persistence of modifications made to detached instances using the `EntityManager.merge()` method:

```
// in the first entity manager
Cat cat = firstEntityManager.find(Cat.class, catId);
Cat potentialMate = new Cat();
firstEntityManager.persist(potentialMate);

// in a higher layer of the application
cat.setMate(potentialMate);

// later, in a new entity manager
secondEntityManager.merge(cat);  // update cat
secondEntityManager.merge(mate); // update mate
```

The `merge()` method merges modifications made to the detached instance into the corresponding managed instance, if any, without consideration of the state of the persistence context. In other words, the merged objects state overrides the persistent entity state in the persistence context, if one is already present. The application should individually `merge()` detached instances reachable from the given detached instance if and only if it wants their state also to be persistent. This can be cascaded to associated entities and collections, using transitive persistence, see *Transitive persistence*.

3.8. Automatic state detection

The merge operation is clever enough to automatically detect whether the merging of the detached instance has to result in an insert or update. In other words, you don't have to worry about passing a new instance (and not a detached instance) to `merge()`, the entity manager will figure this out for you:

```
// In the first entity manager
Cat cat = firstEntityManager.find(Cat.class, catID);

// In a higher layer of the application, detached
Cat mate = new Cat();
cat.setMate(mate);

// Later, in a new entity manager
secondEntityManager.merge(cat);   // update existing state
secondEntityManager.merge(mate);  // save the new instance
```

The usage and semantics of `merge()` seems to be confusing for new users. Firstly, as long as you are not trying to use object state loaded in one entity manager in another new entity manager, you should not need to use `merge()` at all. Some whole applications will never use this method.

Usually `merge()` is used in the following scenario:

- the application loads an object in the first entity manager
- the object is passed up to the presentation layer

- some modifications are made to the object
- the object is passed back down to the business logic layer
- the application persists these modifications by calling `merge()` in a second entity manager

Here is the exact semantic of `merge()`:

- if there is a managed instance with the same identifier currently associated with the persistence context, copy the state of the given object onto the managed instance
- if there is no managed instance currently associated with the persistence context, try to load it from the database, or create a new managed instance
- the managed instance is returned
- the given instance does not become associated with the persistence context, it remains detached and is usually discarded

Merging vs. saveOrUpdate/saveOrUpdateCopy

Merging in JPA is similar to the `saveOrUpdateCopy()` method in native Hibernate. However, it is not the same as the `saveOrUpdate()` method, the given instance is not reattached with the persistence context, but a managed instance is returned by the `merge()` method.

3.9. Deleting managed objects

`EntityManager.remove()` will remove an objects state from the database. Of course, your application might still hold a reference to a deleted object. You can think of `remove()` as making a persistent instance new (aka transient) again. It is not detached, and a merge would result in an insertion.

3.10. Flush the persistence context

From time to time the entity manager will execute the SQL DML statements needed to synchronize the data store with the state of objects held in memory. This process is called flushing.

3.10.1. In a transaction

Flush occurs by default (this is Hibernate specific and not defined by the specification) at the following points:

- before query execution*

- from `javax.persistence.EntityTransaction.commit()`*
- when `EntityManager.flush()` is called*

(*) if a transaction is active

The SQL statements are issued in the following order

- all entity insertions, in the same order the corresponding objects were saved using `EntityManager.persist()`
- all entity updates
- all collection deletions
- all collection element deletions, updates and insertions
- all collection insertions
- all entity deletions, in the same order the corresponding objects were deleted using `EntityManager.remove()`

(Exception: entity instances using application-assigned identifiers are inserted when they are saved.)

Except when you explicitly `flush()`, there are no guarantees about when the entity manager executes the JDBC calls, only the order in which they are executed. However, Hibernate does guarantee that the `Query.getResultList()`/`Query.getSingleResult()` will never return stale data; nor will they return wrong data if executed in an active transaction.

It is possible to change the default behavior so that flush occurs less frequently. The `FlushModeType` for an entity manager defines two different modes: only flush at commit time or flush automatically using the explained routine unless `flush()` is called explicitly.

```
em = emf.createEntityManager();
Transaction tx = em.getTransaction().begin();
em.setFlushMode(FlushModeType.COMMIT); // allow queries to return stale state

Cat izi = em.find(Cat.class, id);
izi.setName(iznizi);

// might return stale data
em.createQuery("from Cat as cat left outer join cat.kittens kitten").getResultList();

// change to izi is not flushed!
...
em.getTransaction().commit(); // flush occurs
```

During flush, an exception might happen (e.g. if a DML operation violates a constraint). TODO: Add link to exception handling.

Hibernate provides more flush modes than the one described in the JPA specification. In particular `FlushMode.MANUAL` for long running conversation. Please refer to the Hibernate core reference documentation for more informations.

3.10.2. Outside a transaction

In an `EXTENDED` persistence context, all read only operations of the entity manager can be executed outside a transaction (`find()`, `getReference()`, `refresh()`, and read queries). Some modifications operations can be executed outside a transaction, but they are queued until the persistence context join a transaction. This is the case of `persist()`, `merge()`, `remove()`. Some operations cannot be called outside a transaction: `flush()`, `lock()`, and update/delete queries.

3.11. Transitive persistence

It is quite cumbersome to save, delete, or reattach individual objects, especially if you deal with a graph of associated objects. A common case is a parent/child relationship. Consider the following example:

If the children in a parent/child relationship would be value typed (e.g. a collection of addresses or strings), their lifecycle would depend on the parent and no further action would be required for convenient "cascading" of state changes. When the parent is persisted, the value-typed child objects are persisted as well, when the parent is removed, the children will be removed, etc. This even works for operations such as the removal of a child from the collection; Hibernate will detect this and, since value-typed objects can't have shared references, remove the child from the database.

Now consider the same scenario with parent and child objects being entities, not value-types (e.g. categories and items, or parent and child cats). Entities have their own lifecycle, support shared references (so removing an entity from the collection does not mean it can be deleted), and there is by default no cascading of state from one entity to any other associated entities. The EJB3 specification does not require persistence by reachability. It supports a more flexible model of transitive persistence, as first seen in Hibernate.

For each basic operation of the entity manager - including `persist()`, `merge()`, `remove()`, `refresh()` - there is a corresponding cascade style. Respectively, the cascade styles are named `PERSIST`, `MERGE`, `REMOVE`, `REFRESH`, `DETACH`. If you want an operation to be cascaded to associated entity (or collection of entities), you must indicate that in the association annotation:

```
@OneToOne(cascade=CascadeType.PERSIST)
```

Cascading options can be combined:

```
@OneToOne(cascade= { CascadeType.PERSIST, CascadeType.REMOVE, CascadeType.REFRESH } )
```

You may even use `CascadeType.ALL` to specify that all operations should be cascaded for a particular association. Remember that by default, no operation is cascaded.

There is an additional cascading mode used to describe orphan deletion (ie an object no longer linked to an owning object should be removed automatically by Hibernate. Use `orphanRemoval=true` on `@OneToOne` or `@OneToMany`. Check Hibernate Annotations's documentation for more information.

Hibernate offers more native cascading options, please refer to the Hibernate Annotations manual and the Hibernate reference guide for more informations.

Recommendations:

- It doesn't usually make sense to enable cascade on a `@ManyToOne` or `@ManyToMany` association. Cascade is often useful for `@OneToOne` and `@OneToMany` associations.
- If the child object's lifespan is bounded by the lifespan of the parent object, make the parent a full lifecycle object by specifying `CascadeType.ALL` and `org.hibernate.annotations.CascadeType.DELETE_ORPHAN` (please refer to the Hibernate reference guide for the semantics of orphan delete)
- Otherwise, you might not need cascade at all. But if you think that you will often be working with the parent and children together in the same transaction, and you want to save yourself some typing, consider using `cascade={PERSIST, MERGE}`. These options can even make sense for a many-to-many association.

3.12. Locking

You can define various levels of locking strategies. A lock can be applied in several ways:

- via the explicit `entityManager.lock()` method
- via lookup methods on `EntityManager`: `find()`, `refresh()`
- on queries: `query.setLockMode()`

You can use various lock approaches:

- `OPTIMISTIC` (previously `READ`): use an optimistic locking scheme where the version number is compared: the version number is compared and has to match before the transaction is committed.
- `OPTIMISTIC_FORCE_INCREMENT` (previously `WRITE`): use an optimistic locking scheme but force a version number increase as well: the version number is compared and has to match before the transaction is committed.
- `PESSIMISTIC_READ`: apply a database-level read lock when the lock operation is requested: roughly concurrent readers are allowed but no writer is allowed.

- `PESSIMISTIC_WRITE`: apply a database-level write lock when the lock operation is requested: roughly no reader nor writer is allowed.

All these locks prevent dirty reads and non-repeatable reads on a given entity. Optimistic locks enforce the lock as late as possible hoping nobody changes the data underneath while pessimistic locks enforce the lock right away and keep it till the transaction is committed.

3.13. Caching

When the second-level cache is activated (see *Section 2.2.1, "Packaging"* and the Hibernate Annotations reference documentation), Hibernate ensures it is used and properly updated. You can however adjust these settings by passing two properties:

- `javax.persistence.cache.retrieveMode` which accepts `CacheRetrieveMode` values
- `javax.persistence.cache.storeMode` which accepts `CacheStoreMode` values

`CacheRetrieveMode` controls how Hibernate accesses information from the second-level cache: `USE` which is the default or `BYPASS` which means ignore the cache. `CacheStoreMode` controls how Hibernate pushes information to the second-level cache: `USE` which is the default and push data in the cache when reading from and writing to the database, `BYPASS` which does not insert new data in the cache (but can invalidate obsolete data) and `REFRESH` which does like default but also force data to be pushed to the cache on database read even if the data is already cached.

You can set these properties:

- on a particular `EntityManager` via the `setProperty` method
- on a query via a query hint (`setHint` method)
- when calling `find()` and `refresh()` and passing the properties in the appropriate `Map`

JPA also introduces an API to interrogate the second-level cache and evict data manually.

```
Cache cache = entityManagerFactory.getCache();

if ( cache.contains(User.class, userId) ) {
   //load it as we don't hit the DB
}

cache.evict(User.class, userId); //manually evict user form the second-level cache
cache.evict(User.class); //evict all users from the second-level cache
cache.evictAll(); //purge the second-level cache entirely
```

3.14. Checking the state of an object

You can check whether an object is managed by the persistence context

```
entityManager.get(Cat.class, catId);
...
boolean isIn = entityManager.contains(cat);
assert isIn;
```

You can also check whether an object, an association or a property is lazy or not. You can do that independently of the underlying persistence provider:

```
PersistenceUtil jpaUtil = Persistence.getPersistenceUtil();
if ( jpaUtil.isLoaded( customer.getAddress() ) {
   //display address if loaded
}
if ( jpaUtil.isLoaded( customer.getOrders ) ) {
   //display orders if loaded
}
if (jpaUtil.isLoaded(customer, "detailedBio") ) {
   //display property detailedBio if loaded
}
```

However, if you have access to the entityManagerFactory, we recommend you to use:

```
PersistenceUnitUtil jpaUtil = entityManager.getEntityManagerFactory().getPersistenceUnitUtil();

Customer customer = entityManager.get( Customer.class, customerId );

if ( jpaUtil.isLoaded( customer.getAddress() ) {
   //display address if loaded
}
if ( jpaUtil.isLoaded( customer.getOrders ) ) {
   //display orders if loaded
}
if (jpaUtil.isLoaded(customer, "detailedBio") ) {
   //display property detailedBio if loaded
}

log.debug( "Customer id {}", jpaUtil.getIdentifier(customer) );
```

The performances are likely to be slightly better and you can get the identifier value from an object (using `getIdentifier()`).

Note

These are roughly the counterpart methods of `Hibernate.isInitialize`.

3.15. Native Hibernate API

You can always fall back to the underlying `Session` API from a given `EntityManager`:

```
Session session = entityManager.unwrap(Session.class);
```

Metamodel

Note

The Metamodel itself is described in *Chapter 5 Metamodel API* of the [*JPA 2 Specification*]. *Chapter 6 Criteria API* of the [*JPA 2 Specification*] describes and shows uses of the metamodel in criteria queries, as does *Chapter 9, Criteria Queries*.

The metamodel is a set of objects that describe your domain model. `javax.persistence.metamodel.Metamodel` acts as a repository of these metamodel objects and provides access to them, and can be obtained from either the `javax.persistence.EntityManagerFactory` or the `javax.persistence.EntityManager` via their `getMetamodel` method.

This metamodel is important in 2 ways. First, it allows providers and frameworks a generic way to deal with an application's domain model. Persistence providers will already have some form of metamodel that they use to describe the domain model being mapped. This API however defines a single, independent access to that existing information. A validation framework, for example, could use this information to understand associations; a marshaling framework might use this information to decide how much of an entity graph to marshal. This usage is beyond the scope of this documentation.

Important

As of today the JPA 2 metamodel does not provide any facility for accessing relational information pertaining to the physical model. It is expected this will be addressed in a future release of the specification.

Second, from an application writer's perspective, it allows very fluent expression of completely type-safe criteria queries, especially the *Static Metamodel* approach. The [*JPA 2 Specification*] defines a number of ways the metamodel can be accessed and used, including the *Static Metamodel* approach, which we will look at later. The *Static Metamodel* approach is wonderful when the code has *a priori knowledge* [http://en.wikipedia.org/wiki/A_priori_and_a_posteriori] of the domain model. *Chapter 9, Criteria Queries* uses this approach exclusively in its examples.

4.1. Static metamodel

A *static metamodel* is a series of classes that "mirror" the entities and embeddables in the domain model and provide static access to the metadata about the mirrored class's attributes. We will exclusively discuss what the [*JPA 2 Specification*] terms a *Canonical Metamodel*:

- For each managed class X in package p, a metamodel class X_ in package p is created.
- The name of the metamodel class is derived from the name of the managed class by appending "_" to the name of the managed class.
- The metamodel class X_ must be annotated with the `javax.persistence.StaticMetamodel`annotation [1]
- If class X extends another class S, where S is the most derived managed class (i.e., entity or mapped superclass) extended by X, then class X_ must extend class S_, where S_ is the metamodel class created for S.
- For every persistent non-collection-valued attribute *y* declared by class X, where the type of *y* is Y, the metamodel class must contain a declaration as follows:

```
public static volatile SingularAttribute<X, Y> y;
```

- For every persistent collection-valued attribute *z* declared by class X, where the element type of *z* is Z, the metamodel class must contain a declaration as follows:
  - if the collection type of *z* is `java.util.Collection`, then

```
public static volatile CollectionAttribute<X, Z> z;
```

  - if the collection type of *z* is `java.util.Set`, then

```
public static volatile SetAttribute<X, Z> z;
```

  - if the collection type of *z* is `java.util.List`, then

```
public static volatile ListAttribute<X, Z> z;
```

  - if the collection type of *z* is `java.util.Map`, then

```
public static volatile MapAttribute<X, K, Z> z;
```

[1] *(from the original)* If the class was generated, the `javax.annotation.Generated` annotation should be used to annotate the class. The use of any other annotations on static metamodel classes is undefined.

where `K` is the type of the key of the map in class `X`

Import statements must be included for the needed javax.persistence.metamodel types as appropriate (e.g., `javax.persistence.metamodel.SingularAttribute`, `javax.persistence.metamodel.CollectionAttribute`, `javax.persistence.metamodel.SetAttribute`, `javax.persistence.metamodel.ListAttribute`, `javax.persistence.metamodel.MapAttribute`) and all classes `X`, `Y`, `Z`, and `K`.

— [*JPA 2 Specification*, section 6.2.1.1, pp 198-199]

Example 4.1. Static metamodel example

For the `Person` entity

```
package org.hibernate.jpa2.metamodel.example;

import java.util.Set;
import javax.persistence.Entity;

@Entity
public class Person {
    @Id private Long id;
    private String name;
    private int age;
    private Address address;
    @OneToMany  private Set<Order> orders;
}
```

The corresponding canonical metamodel class, `Person_` would look like

```
package org.hibernate.jpa2.metamodel.example;

import javax.persistence.metamodel.SingularAttribute;
import javax.persistence.metamodel.SetAttribute;
import javax.persistence.metamodel.StaticMetamodel;

@StaticMetamodel( Person.class )
public class Person_ {
    public static volatile SingularAttribute<Person, Long> id;
    public static volatile SingularAttribute<Person, String> name;
    public static volatile SingularAttribute<Person, Integer> age;
    public static volatile SingularAttribute<Person, Address> address;
    public static volatile SetAttribute<Person, Order> orders;
}
```

Note

These canonical metamodel classes can be generated manually if you wish though it is expected that most developers will prefer use of an *annotation processor* [http://java.sun.com/javase/6/docs/technotes/tools/solaris/javac.html#processing]. Annotation processors themselves are beyond the scope of this document. However, the Hibernate team does develop an annotation processor tool for generating a canonical metamodel. See *Hibernate Metamodel Generator*.

When the Hibernate `EntityManagerFactory` is being built, it will look for a canonical metamodel class for each of the managed typed is knows about and if it finds any it will inject the appropriate metamodel information into them, as outlined in [*JPA 2 Specification*, section 6.2.2, pg 200]

Transactions and Concurrency

The most important point about Hibernate Entity Manager and concurrency control is that it is very easy to understand. Hibernate Entity Manager directly uses JDBC connections and JTA resources without adding any additional locking behavior. We highly recommend you spend some time with the JDBC, ANSI, and transaction isolation specification of your database management system. Hibernate Entity Manager only adds automatic versioning but does not lock objects in memory or change the isolation level of your database transactions. Basically, use Hibernate Entity Manager like you would use direct JDBC (or JTA/CMT) with your database resources.

We start the discussion of concurrency control in Hibernate with the granularity of `EntityManagerFactory`, and `EntityManager`, as well as database transactions and long units of work..

In this chapter, and unless explicitly expressed, we will mix and match the concept of entity manager and persistence context. One is an API and programming object, the other a definition of scope. However, keep in mind the essential difference. A persistence context is usually bound to a JTA transaction in Java EE, and a persistence context starts and ends at transaction boundaries (transaction-scoped) unless you use an extended entity manager. Please refer to *Section 1.2.3, "Persistence context scope"* for more information.

5.1. Entity manager and transaction scopes

A `EntityManagerFactory` is an expensive-to-create, threadsafe object intended to be shared by all application threads. It is created once, usually on application startup.

An `EntityManager` is an inexpensive, non-threadsafe object that should be used once, for a single business process, a single unit of work, and then discarded. An `EntityManager` will not obtain a JDBC `Connection` (or a `Datasource`) unless it is needed, so you may safely open and close an `EntityManager` even if you are not sure that data access will be needed to serve a particular request. (This becomes important as soon as you are implementing some of the following patterns using request interception.)

To complete this picture you also have to think about database transactions. A database transaction has to be as short as possible, to reduce lock contention in the database. Long database transactions will prevent your application from scaling to highly concurrent load.

What is the scope of a unit of work? Can a single Hibernate `EntityManager` span several database transactions or is this a one-to-one relationship of scopes? When should you open and close a `Session` and how do you demarcate the database transaction boundaries?

5.1.1. Unit of work

First, don't use the *entitymanager-per-operation* antipattern, that is, don't open and close an `EntityManager` for every simple database call in a single thread! Of course, the same is true for database transactions. Database calls in an application are made using a planned sequence, they are grouped into atomic units of work. (Note that this also means that auto-commit after every

single SQL statement is useless in an application, this mode is intended for ad-hoc SQL console work.)

The most common pattern in a multi-user client/server application is *entitymanager-per-request*. In this model, a request from the client is send to the server (where the JPA persistence layer runs), a new `EntityManager` is opened, and all database operations are executed in this unit of work. Once the work has been completed (and the response for the client has been prepared), the persistence context is flushed and closed, as well as the entity manager object. You would also use a single database transaction to serve the clients request. The relationship between the two is one-to-one and this model is a perfect fit for many applications.

This is the default JPA persistence model in a Java EE environment (JTA bounded, transaction-scoped persistence context); injected (or looked up) entity managers share the same persistence context for a particular JTA transaction. The beauty of JPA is that you don't have to care about that anymore and just see data access through entity manager and demarcation of transaction scope on session beans as completely orthogonal.

The challenge is the implementation of this (and other) behavior outside an EJB3 container: not only has the `EntityManager` and resource-local transaction to be started and ended correctly, but they also have to be accessible for data access operations. The demarcation of a unit of work is ideally implemented using an interceptor that runs when a request hits the non-EJB3 container server and before the response will be send (i.e. a `ServletFilter` if you are using a standalone servlet container). We recommend to bind the `EntityManager` to the thread that serves the request, using a `ThreadLocal` variable. This allows easy access (like accessing a static variable) in all code that runs in this thread. Depending on the database transaction demarcation mechanism you chose, you might also keep the transaction context in a `ThreadLocal` variable. The implementation patterns for this are known as *ThreadLocal Session* and *Open Session in View* in the Hibernate community. You can easily extend the `HibernateUtil` shown in the Hibernate reference documentation to implement this pattern, you don't need any external software (it's in fact very trivial). Of course, you'd have to find a way to implement an interceptor and set it up in your environment. See the Hibernate website for tips and examples. Once again, remember that your first choice is naturally an EJB3 container - preferably a light and modular one such as JBoss application server.

5.1.2. Long units of work

The entitymanager-per-request pattern is not the only useful concept you can use to design units of work. Many business processes require a whole series of interactions with the user interleaved with database accesses. In web and enterprise applications it is not acceptable for a database transaction to span a user interaction with possibly long waiting time between requests. Consider the following example:

- The first screen of a dialog opens, the data seen by the user has been loaded in a particular `EntityManager` and resource-local transaction. The user is free to modify the detached objects.

- The user clicks "Save" after 5 minutes and expects his modifications to be made persistent; he also expects that he was the only person editing this information and that no conflicting modification can occur.

We call this unit of work, from the point of view of the user, a long running *application transaction*. There are many ways how you can implement this in your application.

A first naive implementation might keep the `EntityManager` and database transaction open during user think time, with locks held in the database to prevent concurrent modification, and to guarantee isolation and atomicity. This is of course an anti-pattern, a pessimistic approach, since lock contention would not allow the application to scale with the number of concurrent users.

Clearly, we have to use several database transactions to implement the application transaction. In this case, maintaining isolation of business processes becomes the partial responsibility of the application tier. A single application transaction usually spans several database transactions. It will be atomic if only one of these database transactions (the last one) stores the updated data, all others simply read data (e.g. in a wizard-style dialog spanning several request/response cycles). This is easier to implement than it might sound, especially if you use JPA entity manager and persistence context features:

- *Automatic Versioning* - An entity manager can do automatic optimistic concurrency control for you, it can automatically detect if a concurrent modification occurred during user think time (usually by comparing version numbers or timestamps when updating the data in the final resource-local transaction).

- *Detached Entities* - If you decide to use the already discussed *entity-per-request* pattern, all loaded instances will be in detached state during user think time. The entity manager allows you to merge the detached (modified) state and persist the modifications, the pattern is called *entitymanager-per-request-with-detached-entities*. Automatic versioning is used to isolate concurrent modifications.

- *Extended Entity Manager* - The Hibernate Entity Manager may be disconnected from the underlying JDBC connection between two client calls and reconnected when a new client request occurs. This pattern is known as *entitymanager-per-application-transaction* and makes even merging unnecessary. An extend persistence context is responsible to collect and retain any modification (persist, merge, remove) made outside a transaction. The next client call made inside an active transaction (typically the last operation of a user conversation) will execute all queued modifications. Automatic versioning is used to isolate concurrent modifications.

Both *entitymanager-per-request-with-detached-objects* and *entitymanager-per-application-transaction* have advantages and disadvantages, we discuss them later in this chapter in the context of optimistic concurrency control.

TODO: This note should probably come later.

5.1.3. Considering object identity

An application may concurrently access the same persistent state in two different persistence contexts. However, an instance of a managed class is never shared between two persistence contexts. Hence there are two different notions of identity:

Database Identity

```
foo.getId().equals( bar.getId() )
```

JVM Identity

```
foo==bar
```

Then for objects attached to a *particular* persistence context (i.e. in the scope of an `EntityManager`) the two notions are equivalent, and JVM identity for database identity is guaranteed by the Hibernate Entity Manager. However, while the application might concurrently access the "same" (persistent identity) business object in two different persistence contexts, the two instances will actually be "different" (JVM identity). Conflicts are resolved using (automatic versioning) at flush/commit time, using an optimistic approach.

This approach leaves Hibernate and the database to worry about concurrency; it also provides the best scalability, since guaranteeing identity in single-threaded units of work only doesn't need expensive locking or other means of synchronization. The application never needs to synchronize on any business object, as long as it sticks to a single thread per `EntityManager`. Within a persistence context, the application may safely use == to compare entities.

However, an application that uses == outside of a persistence context might see unexpected results. This might occur even in some unexpected places, for example, if you put two detached instances into the same `Set`. Both might have the same database identity (i.e. they represent the same row), but JVM identity is by definition not guaranteed for instances in detached state. The developer has to override the `equals()` and `hashCode()` methods in persistent classes and implement his own notion of object equality. There is one caveat: Never use the database identifier to implement equality, use a business key, a combination of unique, usually immutable, attributes. The database identifier will change if a transient entity is made persistent (see the contract of the `persist()` operation). If the transient instance (usually together with detached instances) is held in a `Set`, changing the hashcode breaks the contract of the `Set`. Attributes for good business keys don't have to be as stable as database primary keys, you only have to guarantee stability as long as the objects are in the same `Set`. See the Hibernate website for a more thorough discussion of this issue. Also note that this is not a Hibernate issue, but simply how Java object identity and equality has to be implemented.

5.1.4. Common concurrency control issues

Never use the anti-patterns *entitymanager-per-user-session* or *entitymanager-per-application* (of course, there are rare exceptions to this rule, e.g. entitymanager-per-application might be acceptable in a desktop application, with manual flushing of the persistence context). Note that

some of the following issues might also appear with the recommended patterns, make sure you understand the implications before making a design decision:

- An entity manager is not thread-safe. Things which are supposed to work concurrently, like HTTP requests, session beans, or Swing workers, will cause race conditions if an `EntityManager` instance would be shared. If you keep your Hibernate `EntityManager` in your `HttpSession` (discussed later), you should consider synchronizing access to your Http session. Otherwise, a user that clicks reload fast enough may use the same `EntityManager` in two concurrently running threads. You will very likely have provisions for this case already in place, for other non-threadsafe but session-scoped objects.
- An exception thrown by the Entity Manager means you have to rollback your database transaction and close the `EntityManager` immediately (discussed later in more detail). If your `EntityManager` is bound to the application, you have to stop the application. Rolling back the database transaction doesn't put your business objects back into the state they were at the start of the transaction. This means the database state and the business objects do get out of sync. Usually this is not a problem, because exceptions are not recoverable and you have to start over your unit of work after rollback anyway.
- The persistence context caches every object that is in managed state (watched and checked for dirty state by Hibernate). This means it grows endlessly until you get an `OutOfMemoryException`, if you keep it open for a long time or simply load too much data. One solution for this is some kind batch processing with regular flushing of the persistence context, but you should consider using a database stored procedure if you need mass data operations. Some solutions for this problem are shown in *Chapter 7, Batch processing*. Keeping a persistence context open for the duration of a user session also means a high probability of stale data, which you have to know about and control appropriately.

5.2. Database transaction demarcation

Database (or system) transaction boundaries are always necessary. No communication with the database can occur outside of a database transaction (this seems to confuse many developers who are used to the auto-commit mode). Always use clear transaction boundaries, even for read-only operations. Depending on your isolation level and database capabilities this might not be required but there is no downside if you always demarcate transactions explicitly. You'll have to do operations outside a transaction, though, when you'll need to retain modifications in an `EXTENDED` persistence context.

A JPA application can run in non-managed (i.e. standalone, simple Web- or Swing applications) and managed Java EE environments. In a non-managed environment, an `EntityManagerFactory` is usually responsible for its own database connection pool. The application developer has to manually set transaction boundaries, in other words, begin, commit, or rollback database transactions itself. A managed environment usually provides container-managed transactions, with the transaction assembly defined declaratively through annotations of EJB session beans, for example. Programmatic transaction demarcation is then no longer necessary, even flushing the `EntityManager` is done automatically.

Usually, ending a unit of work involves four distinct phases:

- commit the (resource-local or JTA) transaction (this automatically flushes the entity manager and persistence context)
- close the entity manager (if using an application-managed entity manager)
- handle exceptions

We'll now have a closer look at transaction demarcation and exception handling in both managed- and non-managed environments.

5.2.1. Non-managed environment

If an JPA persistence layer runs in a non-managed environment, database connections are usually handled by Hibernate's pooling mechanism behind the scenes. The common entity manager and transaction handling idiom looks like this:

```
// Non-managed environment idiom
EntityManager em = emf.createEntityManager();
EntityTransaction tx = null;
try {
    tx = em.getTransaction();
    tx.begin();

    // do some work
    ...

    tx.commit();
}
catch (RuntimeException e) {
    if ( tx != null && tx.isActive() ) tx.rollback();
    throw e; // or display error message
}
finally {
    em.close();
}
```

You don't have to `flush()` the `EntityManager` explicitly - the call to `commit()` automatically triggers the synchronization.

A call to `close()` marks the end of an `EntityManager`. The main implication of `close()` is the release of resources - make sure you always close and never outside of guaranteed finally block.

You will very likely never see this idiom in business code in a normal application; fatal (system) exceptions should always be caught at the "top". In other words, the code that executes entity manager calls (in the persistence layer) and the code that handles `RuntimeException` (and usually can only clean up and exit) are in different layers. This can be a challenge to design yourself and you should use J2EE/EJB container services whenever they are available. Exception handling is discussed later in this chapter.

5.2.1.1. EntityTransaction

In a JTA environment, you don't need any extra API to interact with the transaction in your environment. Simply use transaction declaration or the JTA APIs.

If you are using a `RESOURCE_LOCAL` entity manager, you need to demarcate your transaction boundaries through the `EntityTransaction` API. You can get an `EntityTransaction` through `entityManager.getTransaction()`. This `EntityTransaction` API provides the regular `begin()`, `commit()`, `rollback()` and `isActive()` methods. It also provide a way to mark a transaction as rollback only, ie force the transaction to rollback. This is very similar to the JTA operation `setRollbackOnly()`. When a `commit()` operation fail and/or if the transaction is marked as `setRollbackOnly()`, the `commit()` method will try to rollback the transaction and raise a `javax.transaction.RollbackException`.

In a `JTA` entity manager, `entityManager.getTransaction()` calls are not permitted.

5.2.2. Using JTA

If your persistence layer runs in an application server (e.g. behind EJB3 session beans), every datasource connection obtained internally by the entity manager will automatically be part of the global JTA transaction. Hibernate offers two strategies for this integration.

If you use bean-managed transactions (BMT), the code will look like this:

```
// BMT idiom
@Resource public UserTransaction utx;
@Resource public EntityManagerFactory factory;

public void doBusiness() {
    EntityManager em = factory.createEntityManager();
    try {

    // do some work
    ...

    utx.commit();
}
catch (RuntimeException e) {
    if (utx != null) utx.rollback();
    throw e; // or display error message
}
finally {
    em.close();
}
```

With Container Managed Transactions (CMT) in an EJB3 container, transaction demarcation is done in session bean annotations or deployment descriptors, not programatically. The `EntityManager` will automatically be flushed on transaction completion (and if you have injected or lookup the `EntityManager`, it will be also closed automatically). If an exception occurs during the `EntityManager` use, transaction rollback occurs automatically if you don't catch the exception.

Since `EntityManager` exceptions are `RuntimeExceptions` they will rollback the transaction as per the EJB specification (system exception vs. application exception).

It is important to let Hibernate EntityManager define the `hibernate.transaction.factory_class` (ie not overriding this value). Remember to also set `org.hibernate.transaction.manager_lookup_class`.

If you work in a CMT environment, you might also want to use the same entity manager in different parts of your code. Typically, in a non-managed environment you would use a `ThreadLocal` variable to hold the entity manager, but a single EJB request might execute in different threads (e.g. session bean calling another session bean). The EJB3 container takes care of the persistence context propagation for you. Either using injection or lookup, the EJB3 container will return an entity manager with the same persistence context bound to the JTA context if any, or create a new one and bind it (see *Section 1.2.4, "Persistence context propagation"* .)

Our entity manager/transaction management idiom for CMT and EJB3 container-use is reduced to this:

```
//CMT idiom through injection
@PersistenceContext(name="sample") EntityManager em;
```

Or this if you use Java Context and Dependency Injection (CDI).

```
@Inject EntityManager em;
```

In other words, all you have to do in a managed environment is to inject the `EntityManager`, do your data access work, and leave the rest to the container. Transaction boundaries are set declaratively in the annotations or deployment descriptors of your session beans. The lifecycle of the entity manager and persistence context is completely managed by the container.

Due to a silly limitation of the JTA spec, it is not possible for Hibernate to automatically clean up any unclosed `ScrollableResults` or `Iterator` instances returned by `scroll()` or `iterate()`. You *must* release the underlying database cursor by calling `ScrollableResults.close()` or `Hibernate.close(Iterator)` explicitly from a `finally` block. (Of course, most applications can easily avoid using `scroll()` or `iterate()` at all from the CMT code.)

5.2.3. Exception handling

If the `EntityManager` throws an exception (including any `SQLException`), you should immediately rollback the database transaction, call `EntityManager.close()` (if `createEntityManager()` has been called) and discard the `EntityManager` instance. Certain methods of `EntityManager` will *not* leave the persistence context in a consistent state. No exception thrown by an entity manager can be treated as recoverable. Ensure that the `EntityManager` will be closed by calling `close()` in a `finally` block. Note that a container managed entity manager will do that for you. You just have to let the RuntimeException propagate up to the container.

The Hibernate entity manager generally raises exceptions which encapsulate the Hibernate core exception. Common exceptions raised by the `EntityManager` API are

- `IllegalArgumentException`: something wrong happen
- `EntityNotFoundException`: an entity was expected but none match the requirement
- `NonUniqueResultException`: more than one entity is found when calling `getSingleResult()`
- NoResultException: when `getSingleResult()` does not find any matching entity
- `EntityExistsException`: an existing entity is passed to `persist()`
- `TransactionRequiredException`: this operation has to be in a transaction
- `IllegalStateException`: the entity manager is used in a wrong way
- `RollbackException`: a failure happens during `commit()`
- `QueryTimeoutException`: the query takes longer than the specified timeout (see `javax.persistence.query.timeout` - this property is a hint and might not be followed)
- `PessimisticLockException`: when a lock cannot be acquired
- `OptimisticLockException`: an optimistic lock is failing
- `LockTimeoutException`: when a lock takes longer than the expected time to be acquired (`javax.persistence.lock.timeout` in milliseconds)
- `TransactionRequiredException`: an operation requiring a transaction is executed outside of a transaction

The `HibernateException`, which wraps most of the errors that can occur in a Hibernate persistence layer, is an unchecked exception. Note that Hibernate might also throw other unchecked exceptions which are not a `HibernateException`. These are, again, not recoverable and appropriate action should be taken.

Hibernate wraps `SQLException`s thrown while interacting with the database in a `JDBCException`. In fact, Hibernate will attempt to convert the exception into a more meaningful subclass of `JDBCException`. The underlying `SQLException` is always available via `JDBCException.getCause()`. Hibernate converts the `SQLException` into an appropriate `JDBCException` subclass using the `SQLExceptionConverter` attached to the `SessionFactory`. By default, the `SQLExceptionConverter` is defined by the configured dialect; however, it is also possible to plug in a custom implementation (see the javadocs for the `SQLExceptionConverterFactory` class for details). The standard `JDBCException` subtypes are:

- `JDBCConnectionException` - indicates an error with the underlying JDBC communication.
- `SQLGrammarException` - indicates a grammar or syntax problem with the issued SQL.
- `ConstraintViolationException` - indicates some form of integrity constraint violation.

- `LockAcquisitionException` - indicates an error acquiring a lock level necessary to perform the requested operation.
- `GenericJDBCException` - a generic exception which did not fall into any of the other categories.

5.3. EXTENDED Persistence Context

All application managed entity manager and container managed persistence contexts defined as such are `EXTENDED`. This means that the persistence context type goes beyond the transaction life cycle. We should then understand what happens to operations made outside the scope of a transaction.

In an `EXTENDED` persistence context, all read only operations of the entity manager can be executed outside a transaction (`find()`, `getReference()`, `refresh()`, `detach()` and read queries). Some modifications operations can be executed outside a transaction, but they are queued until the persistence context join a transaction: this is the case of `persist()`, `merge()`, `remove()`. Some operations cannot be called outside a transaction: `flush()`, `lock()`, and update/delete queries.

5.3.1. Container Managed Entity Manager

When using an `EXTENDED` persistence context with a container managed entity manager, the lifecycle of the persistence context is binded to the lifecycle of the Stateful Session Bean. Plus if the entity manager is created outside a transaction, modifications operations (persist, merge, remove) are queued in the persistence context and not executed to the database.

When a method of the stateful session bean involved or starting a transaction is later called, the entity manager join the transaction. All queued operation will then be executed to synchronize the persistence context.

This is perfect to implement the `entitymanager-per-conversation` pattern. A stateful session bean represents the conversation implementation. All intermediate conversation work will be processed in methods not involving transaction. The end of the conversation will be processed inside a `JTA` transaction. Hence all queued operations will be executed to the database and committed. If you are interested in the notion of conversation inside your application, have a look at JBoss Seam. JBoss Seam emphasizes the concept of conversation and entity manager lifecycle and bind EJB3 and JSF together.

5.3.2. Application Managed Entity Manager

Application-managed entity manager are always `EXTENDED`. When you create an entity manager inside a transaction, the entity manager automatically join the current transaction. If the entity manager is created outside a transaction, the entity manager will queue the modification operations. When

- `entityManager.joinTransaction()` is called when a JTA transaction is active for a `JTA` entity manager
- `entityManager.getTransaction().begin()` is called for a `RESOURCE_LOCAL` entity manager

the entity manager join the transaction and all the queued operations will then be executed to synchronize the persistence context.

It is not legal to call `entityManager.joinTransaction()` if no JTA transaction is involved.

5.4. Optimistic concurrency control

The only approach that is consistent with high concurrency and high scalability is optimistic concurrency control with versioning. Version checking uses version numbers, or timestamps, to detect conflicting updates (and to prevent lost updates). Hibernate provides for three possible approaches to writing application code that uses optimistic concurrency. The use cases we show are in the context of long application transactions but version checking also has the benefit of preventing lost updates in single database transactions.

5.4.1. Application version checking

In an implementation without much help from the persistence mechanism, each interaction with the database occurs in a new `EntityManager` and the developer is responsible for reloading all persistent instances from the database before manipulating them. This approach forces the application to carry out its own version checking to ensure application transaction isolation. This approach is the least efficient in terms of database access. It is the approach most similar to EJB2 entities:

```
// foo is an instance loaded by a previous entity manager
em = factory.createEntityManager();
EntityTransaction t = em.getTransaction();
t.begin();
int oldVersion = foo.getVersion();
Foo dbFoo = em.find( foo.getClass(), foo.getKey() ); // load the current state
if ( dbFoo.getVersion()!=foo.getVersion ) throw new StaleObjectStateException();
dbFoo.setProperty("bar");
t.commit();
em.close();
```

The `version` property is mapped using `@Version`, and the entity manager will automatically increment it during flush if the entity is dirty.

Of course, if you are operating in a low-data-concurrency environment and don't require version checking, you may use this approach and just skip the version check. In that case, *last commit wins* will be the default strategy for your long application transactions. Keep in mind that this might confuse the users of the application, as they might experience lost updates without error messages or a chance to merge conflicting changes.

Clearly, manual version checking is only feasible in very trivial circumstances and not practical for most applications. Often not only single instances, but complete graphs of modified objects have to be checked. Hibernate offers automatic version checking with either detached instances or an extended entity manager and persistence context as the design paradigm.

5.4.2. Extended entity manager and automatic versioning

A single persistence context is used for the whole application transaction. The entity manager checks instance versions at flush time, throwing an exception if concurrent modification is detected. It's up to the developer to catch and handle this exception (common options are the opportunity for the user to merge his changes or to restart the business process with non-stale data).

In an `EXTENDED` persistence context, all operations made outside an active transaction are queued. The `EXTENDED` persistence context is flushed when executed in an active transaction (at worse at commit time).

The `Entity Manager` is disconnected from any underlying JDBC connection when waiting for user interaction. In an application-managed extended entity manager, this occurs automatically at transaction completion. In a stateful session bean holding a container-managed extended entity manager (i.e. a SFSB annotated with `@PersistenceContext(EXTENDED)`), this occurs transparently as well. This approach is the most efficient in terms of database access. The application need not concern itself with version checking or with merging detached instances, nor does it have to reload instances in every database transaction. For those who might be concerned by the number of connections opened and closed, remember that the connection provider should be a connection pool, so there is no performance impact. The following examples show the idiom in a non-managed environment:

```
// foo is an instance loaded earlier by the extended entity manager
em.getTransaction.begin(); // new connection to data store is obtained and tx started
foo.setProperty("bar");
em.getTransaction().commit();  // End tx, flush and check version, disconnect
```

The `foo` object still knows which `persistence context` it was loaded in. With `getTransaction.begin();` the entity manager obtains a new connection and resumes the persistence context. The method `getTransaction().commit()` will not only flush and check versions, but also disconnects the entity manager from the JDBC connection and return the connection to the pool.

This pattern is problematic if the persistence context is too big to be stored during user think time, and if you don't know where to store it. E.g. the `HttpSession` should be kept as small as possible. As the persistence context is also the (mandatory) first-level cache and contains all loaded objects, we can probably use this strategy only for a few request/response cycles. This is indeed recommended, as the persistence context will soon also have stale data.

It is up to you where you store the extended entity manager during requests, inside an EJB3 container you simply use a stateful session bean as described above. Don't transfer it to the web layer (or even serialize it to a separate tier) to store it in the `HttpSession`. In a non-managed, two-tiered environment the `HttpSession` might indeed be the right place to store it.

5.4.3. Detached objects and automatic versioning

With this paradigm, each interaction with the data store occurs in a new persistence context. However, the same persistent instances are reused for each interaction with the database. The application manipulates the state of detached instances originally loaded in another persistence context and then merges the changes using `EntityManager.merge()`:

```
// foo is an instance loaded by a non-extended entity manager
foo.setProperty("bar");
entityManager = factory.createEntityManager();
entityManager.getTransaction().begin();
managedFoo = session.merge(foo); // discard foo and from now on use managedFoo
entityManager.getTransaction().commit();
entityManager.close();
```

Again, the entity manager will check instance versions during flush, throwing an exception if conflicting updates occurred.

Entity listeners and Callback methods

6.1. Definition

It is often useful for the application to react to certain events that occur inside the persistence mechanism. This allows the implementation of certain kinds of generic functionality, and extension of built-in functionality. The JPA specification provides two related mechanisms for this purpose.

A method of the entity may be designated as a callback method to receive notification of a particular entity life cycle event. Callbacks methods are annotated by a callback annotation. You can also define an entity listener class to be used instead of the callback methods defined directly inside the entity class. An entity listener is a stateless class with a no-arg constructor. An entity listener is defined by annotating the entity class with the `@EntityListeners` annotation:

```
@Entity
@EntityListeners(class=Audit.class)
public class Cat {
    @Id private Integer id;
    private String name;
    private Calendar dateOfBirth;
    @Transient private int age;
    private Date lastUpdate;
    //getters and setters

    /**
     * Set my transient property at load time based on a calculation,
     * note that a native Hibernate formula mapping is better for this purpose.
     */
    @PostLoad
    public void calculateAge() {
        Calendar birth = new GregorianCalendar();
        birth.setTime(dateOfBirth);
        Calendar now = new GregorianCalendar();
        now.setTime( new Date() );
        int adjust = 0;
        if ( now.get(Calendar.DAY_OF_YEAR) - birth.get(Calendar.DAY_OF_YEAR) < 0) {
            adjust = -1;
        }
        age = now.get(Calendar.YEAR) - birth.get(Calendar.YEAR) + adjust;
    }
}

public class LastUpdateListener {
    /**
     * automatic property set before any database persistence
     */
    @PreUpdate
    @PrePersist
    public void setLastUpdate(Cat o) {
        o.setLastUpdate( new Date() );
```

```
    }
}
```

The same callback method or entity listener method can be annotated with more than one callback annotation. For a given entity, you cannot have two methods being annotated by the same callback annotation whether it is a callback method or an entity listener method. A callback method is a no-arg method with no return type and any arbitrary name. An entity listener has the signature `void <METHOD>(Object)` where Object is of the actual entity type (note that Hibernate Entity Manager relaxed this constraint and allows `Object` of `java.lang.Object` type (allowing sharing of listeners across several entities.)

A callback method can raise a `RuntimeException`. The current transaction, if any, must be rolled back. The following callbacks are defined:

Table 6.1. Callbacks

Type	Description
@PrePersist	Executed before the entity manager persist operation is actually executed or cascaded. This call is synchronous with the persist operation.
@PreRemove	Executed before the entity manager remove operation is actually executed or cascaded. This call is synchronous with the remove operation.
@PostPersist	Executed after the entity manager persist operation is actually executed or cascaded. This call is invoked after the database INSERT is executed.
@PostRemove	Executed after the entity manager remove operation is actually executed or cascaded. This call is synchronous with the remove operation.
@PreUpdate	Executed before the database UPDATE operation.
@PostUpdate	Executed after the database UPDATE operation.
@PostLoad	Executed after an entity has been loaded into the current persistence context or an entity has been refreshed.

A callback method must not invoke `EntityManager` or `Query` methods!

6.2. Callbacks and listeners inheritance

You can define several entity listeners per entity at different level of the hierarchy. You can also define several callbacks at different level of the hierarchy. But you cannot define two listeners for the same event in the same entity or the same entity listener.

When an event is raised, the listeners are executed in this order:

- `@EntityListeners` for a given entity or superclass in the array order
- Entity listeners for the superclasses (highest first)
- Entity Listeners for the entity
- Callbacks of the superclasses (highest first)
- Callbacks of the entity

You can stop the entity listeners inheritance by using the `@ExcludeSuperclassListeners`, all superclasses `@EntityListeners` will then be ignored.

6.3. XML definition

The JPA specification allows annotation overriding through JPA deployment descriptors. There is also an additional feature that can be useful: default event listeners.

```
<?xml version="1.0" encoding="UTF-8"?>

<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm orm_2_0.xsd"
                 version="2.0"
        >
    <persistence-unit-metadata>
        <persistence-unit-defaults>
            <entity-listeners>
             <entity-listener class="org.hibernate.ejb.test.pack.defaultpar.IncrementListener">
                    <pre-persist method-name="increment"/>
                </entity-listener>
            </entity-listeners>
        </persistence-unit-defaults>
    </persistence-unit-metadata>
    <package>org.hibernate.ejb.test.pack.defaultpar</package>
    <entity class="ApplicationServer">
        <entity-listeners>
            <entity-listener class="OtherIncrementListener">
                <pre-persist method-name="increment"/>
            </entity-listener>
        </entity-listeners>


        <pre-persist method-name="calculate"/>
```

```
    </entity>
</entity-mappings>
```

You can override entity listeners on a given entity. An entity listener correspond to a given class and one or several event fire a given method call. You can also define event on the entity itself to describe the callbacks.

Last but not least, you can define some default entity listeners that will apply first on the entity listener stack of all the mapped entities of a given persistence unit. If you don't want an entity to inherit the default listeners, you can use `@ExcludeDefaultListeners` (or <exclude-default-listeners/>).

Batch processing

Batch processing has traditionally been difficult in full object/relational mapping. ORM is all about object state management, which implies that object state is available in memory. However, Hibernate has some features to optimize batch processing which are discussed in the Hibernate reference guide, however, EJB3 persistence differs slightly.

7.1. Bulk update/delete

As already discussed, automatic and transparent object/relational mapping is concerned with the management of object state. This implies that the object state is available in memory, hence updating or deleting (using SQL `UPDATE` and `DELETE`) data directly in the database will not affect in-memory state. However, Hibernate provides methods for bulk SQL-style `UPDATE` and `DELETE` statement execution which are performed through JP-QL (*Chapter 8, JP-QL: The Object Query Language*).

The pseudo-syntax for `UPDATE` and `DELETE` statements is: `( UPDATE | DELETE ) FROM? ClassName (WHERE WHERE_CONDITIONS)?`. Note that:

- In the from-clause, the FROM keyword is optional.
- There can only be a single class named in the from-clause, and it *cannot* have an alias (this is a current Hibernate limitation and will be removed soon).
- No joins (either implicit or explicit) can be specified in a bulk JP-QL query. Sub-queries may be used in the where-clause.
- The where-clause is also optional.

As an example, to execute an JP-QL `UPDATE`, use the `Query.executeUpdate()` method:

```
EntityManager entityManager = entityManagerFactory.createEntityManager();
entityManager.getTransaction().begin();

String jpqlUpdate = "update Customer set name = :newName where name = :oldName"
int updatedEntities = entityManager.createQuery( jpqlUpdate )
                            .setParameter( "newName", newName )
                            .setParameter( "oldName", oldName )
                            .executeUpdate();
entityManager.getTransaction().commit();
entityManager.close();
```

To execute an JP-QL `DELETE`, use the same `Query.executeUpdate()` method (the method is named for those familiar with JDBC's `PreparedStatement.executeUpdate()`):

```
EntityManager entityManager = entityManagerFactory.createEntityManager();
entityManager.getTransaction().begin();
```

```
String hqlDelete = "delete Customer where name = :oldName";
int deletedEntities = entityManager.createQuery( hqlDelete )
                            .setParameter( "oldName", oldName )
                            .executeUpdate();
entityManager.getTransaction().commit();
entityManager.close();
```

The `int` value returned by the `Query.executeUpdate()` method indicate the number of entities effected by the operation. This may or may not correlate with the number of rows effected in the database. A JP-QL bulk operation might result in multiple actual SQL statements being executed, for joined-subclass, for example. The returned number indicates the number of actual entities affected by the statement. Going back to the example of joined-subclass, a delete against one of the subclasses may actually result in deletes against not just the table to which that subclass is mapped, but also the "root" table and potentially joined-subclass tables further down the inheritance hierarchy.

JP-QL: The Object Query Language

The Java Persistence Query Language (JP-QL) has been heavily inspired by HQL, the native Hibernate Query Language. Both are therefore very close to SQL, but portable and independent of the database schema. People familiar with HQL shouldn't have any problem using JP-QL. In fact HQL is a strict superset of JP-QL and you use the same query API for both types of queries. Portable JPA applications however should stick to JP-QL.

Note

For a type-safe approach to query, we highly recommend you to use the Criteria query, see *Chapter 9, Criteria Queries*.

8.1. Case Sensitivity

Queries are case-insensitive, except for names of Java classes and properties. So `SeLeCT` is the same as `sELEct` is the same as `SELECT` but `org.hibernate.eg.FOO` is not `org.hibernate.eg.Foo` and `foo.barSet` is not `foo.BARSET`.

This manual uses lowercase JP-QL keywords. Some users find queries with uppercase keywords more readable, but we find this convention ugly when embedded in Java code.

8.2. The from clause

The simplest possible JP-QL query is of the form:

```
select c from eg.Cat c
```

which simply returns all instances of the class `eg.Cat`. Unlike HQL, the select clause is not optional in JP-QL. We don't usually need to qualify the class name, since the entity name defaults to the unqualified class name (`@Entity`). So we almost always just write:

```
select c from Cat c
```

As you may have noticed you can assign aliases to classes, the `as` keywork is optional. An alias allows you to refer to `Cat` in other parts of the query.

```
select cat from Cat as cat
```

Multiple classes may appear, resulting in a cartesian product or "cross" join.

```
select from, param from Formula as form, Parameter as param
```

It is considered good practice to name query aliases using an initial lowercase, consistent with Java naming standards for local variables (eg. `domesticCat`).

8.3. Associations and joins

You may also assign aliases to associated entities, or even to elements of a collection of values, using a `join`.

```
select cat, mate, kitten from Cat as cat
    inner join cat.mate as mate
    left outer join cat.kittens as kitten
```

```
select cat from Cat as cat left join cat.mate.kittens as kittens
```

The supported join types are borrowed from ANSI SQL

- `inner join`
- `left outer join`

The `inner join`, `left outer join` constructs may be abbreviated.

```
select cat, mate, kitten from Cat as cat
    join cat.mate as mate
    left join cat.kittens as kitten
```

In addition, a "fetch" join allows associations or collections of values to be initialized along with their parent objects, using a single select. This is particularly useful in the case of a collection. It effectively overrides the fetching options in the associations and collection mapping metadata. See the Performance chapter of the Hibernate reference guide for more information.

```
select cat from Cat as cat
    inner join fetch cat.mate
    left join fetch cat.kittens
```

A fetch join does not usually need to assign an alias, because the associated objects should not be used in the `where` clause (or any other clause). Also, the associated objects are not returned directly in the query results. Instead, they may be accessed via the parent object. The only reason we might need an alias is if we are recursively join fetching a further collection:

```
select cat from Cat as cat
    inner join fetch cat.mate
    left join fetch cat.kittens child
    left join fetch child.kittens
```

Note that the `fetch` construct may not be used in queries called using `scroll()` or `iterate()`. Nor should `fetch` be used together with `setMaxResults()` or `setFirstResult()`. It is possible to create a cartesian product by join fetching more than one collection in a query (as in the example above), be careful the result of this product isn't bigger than you expect. Join fetching multiple collection roles gives unexpected results for bag mappings as it is impossible for Hibernate to differentiate legit duplicates of a given bag from artificial duplicates created by the multi-table cartesian product.

If you are using property-level lazy fetching (with bytecode instrumentation), it is possible to force Hibernate to fetch the lazy properties immediately (in the first query) using `fetch all properties`. This is Hibernate specific option:

```
select doc from Document doc fetch all properties order by doc.name
```

```
select doc from Document doc fetch all properties where lower(doc.name) like '%cats%'
```

8.4. The select clause

The `select` clause picks which objects and properties to return in the query result set. Consider:

```
select mate
from Cat as cat
    inner join cat.mate as mate
```

The query will select `mate`s of other `Cat`s. Actually, you may express this query more compactly as:

```
select cat.mate from Cat cat
```

Queries may return properties of any value type including properties of component type:

```
select cat.name from DomesticCat cat
where cat.name like 'fri%'
```

```
select cust.name.firstName from Customer as cust
```

Queries may return multiple objects and/or properties as an array of type `Object[]`,

```
select mother, offspr, mate.name
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

or as a `List` (HQL specific feature)

```
select new list(mother, offspr, mate.name)
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

or as an actual type-safe Java object (often called a view object),

```
select new Family(mother, mate, offspr)
from DomesticCat as mother
    join mother.mate as mate
    left join mother.kittens as offspr
```

assuming that the class `Family` has an appropriate constructor.

You may assign aliases to selected expressions using `as`:

```
select max(bodyWeight) as max, min(bodyWeight) as min, count(*) as n
from Cat cat
```

This is most useful when used together with `select new map` (HQL specific feature):

```
select new map( max(bodyWeight) as max, min(bodyWeight) as min, count(*) as n )
from Cat cat
```

This query returns a `Map` from aliases to selected values.

8.5. Aggregate functions

HQL queries may even return the results of aggregate functions on properties:

```
select avg(cat.weight), sum(cat.weight), max(cat.weight), count(cat)
from Cat cat
```

The supported aggregate functions are

- `avg(...), avg(distinct ...), sum(...), sum(distinct ...), min(...), max(...)`
- `count(*)`
- `count(...), count(distinct ...), count(all...)`

You may use arithmetic operators, concatenation, and recognized SQL functions in the select clause (dpending on configured dialect, HQL specific feature):

```
select cat.weight + sum(kitten.weight)
from Cat cat
    join cat.kittens kitten
group by cat.id, cat.weight
```

```
select firstName||' '||initial||' '||upper(lastName) from Person
```

The `distinct` and `all` keywords may be used and have the same semantics as in SQL.

```
select distinct cat.name from Cat cat

select count(distinct cat.name), count(cat) from Cat cat
```

8.6. Polymorphic queries

A query like:

```
select cat from Cat as cat
```

returns instances not only of `Cat`, but also of subclasses like `DomesticCat`. Hibernate queries may name *any* Java class or interface in the `from` clause (portable JP-QL queries should only name mapped entities). The query will return instances of all persistent classes that extend that class or implement the interface. The following query would return all persistent objects:

```
from java.lang.Object o // HQL only
```

The interface `Named` might be implemented by various persistent classes:

```
from Named n, Named m where n.name = m.name // HQL only
```

Note that these last two queries will require more than one SQL `SELECT`. This means that the `order by` clause does not correctly order the whole result set. (It also means you can't call these queries using `Query.scroll()`.)

8.7. The where clause

The `where` clause allows you to narrow the list of instances returned. If no alias exists, you may refer to properties by name:

```
select cat from Cat cat where cat.name='Fritz'
```

returns instances of `Cat` named 'Fritz'.

```
select foo
from Foo foo, Bar bar
where foo.startDate = bar.date
```

will return all instances of `Foo` for which there exists an instance of `bar` with a `date` property equal to the `startDate` property of the `Foo`. Compound path expressions make the `where` clause extremely powerful. Consider:

```
select cat from Cat cat where cat.mate.name is not null
```

This query translates to an SQL query with a table (inner) join. If you were to write something like

```
select foo from Foo foo
where foo.bar.baz.customer.address.city is not null
```

you would end up with a query that would require four table joins in SQL.

The = operator may be used to compare not only properties, but also instances:

```
select cat, rival from Cat cat, Cat rival where cat.mate = rival.mate
```

```
select cat, mate
from Cat cat, Cat mate
```

```
where cat.mate = mate
```

The special property (lowercase) `id` may be used to reference the unique identifier of an object. (You may also use its mapped identifer property name.). Note that this keyword is specific to HQL.

```
select cat from Cat as cat where cat.id = 123

select cat from Cat as cat where cat.mate.id = 69
```

The second query is efficient. No table join is required!

Properties of composite identifiers may also be used. Suppose `Person` has a composite identifier consisting of `country` and `medicareNumber`.

```
select person from bank.Person person
where person.id.country = 'AU'
    and person.id.medicareNumber = 123456
```

```
select account from bank.Account account
where account.owner.id.country = 'AU'
    and account.owner.id.medicareNumber = 123456
```

Once again, the second query requires no table join.

Likewise, the special property `class` accesses the discriminator value of an instance in the case of polymorphic persistence. A Java class name embedded in the where clause will be translated to its discriminator value. Once again, this is specific to HQL.

```
select cat from Cat cat where cat.class = DomesticCat
```

You may also specify properties of components or composite user types (and of components of components, etc). Never try to use a path-expression that ends in a property of component type (as opposed to a property of a component). For example, if `store.owner` is an entity with a component `address`

```
store.owner.address.city    // okay
store.owner.address         // error!
```

An "any" type has the special properties `id` and `class`, allowing us to express a join in the following way (where `AuditLog.item` is a property mapped with `<any>`). `Any` is specific to Hibernate

```
from AuditLog log, Payment payment
where log.item.class = 'Payment' and log.item.id = payment.id
```

Notice that `log.item.class` and `payment.class` would refer to the values of completely different database columns in the above query.

8.8. Expressions

Expressions allowed in the `where` clause include most of the kind of things you could write in SQL:

- mathematical operators `+, -, *, /`
- binary comparison operators `=, >=, <=, <>, !=, like`
- logical operations `and, or, not`
- Parentheses `( )`, indicating grouping
- `in`, `not in`, `between`, `is null`, `is not null`, `is empty`, `is not empty`, `member of` and `not member of`
- `exists`, `all`, `any`, `some` (taking subqueries)
- "Simple" case, `case ... when ... then ... else ... end`, and "searched" case, `case when ... then ... else ... end`
- string concatenation `...||...` or `concat(...,...)` `(use concat() for portable JP-QL queries)`
- `current_date()`, `current_time()`, `current_timestamp()`
- `second(...)`, `minute(...)`, `hour(...)`, `day(...)`, `month(...)`, `year(...)`, (specific to HQL)
- Any function or operator: `substring(), trim(), lower(), upper(), length(), locate(), abs(), sqrt(), bit_length()`
- `coalesce()` and `nullif()`
- `TYPE ... in ...`, where the first argument is an identifier variable and the second argument is the subclass to restrict polymorphism to (or a list of subclasses surrounded by parenthesis)
- `cast(... as ...)`, where the second argument is the name of a Hibernate type, and `extract(... from ...)` if ANSI `cast()` and `extract()` is supported by the underlying database
- Any database-supported SQL scalar function like `sign()`, `trunc()`, `rtrim()`, `sin()`
- JDBC IN parameters `?`
- named parameters `:name`, `:start_date`, `:x1`
- SQL literals `'foo'`, `69`, `'1970-01-01 10:00:01.0'`
- JDBC escape syntax for dates (dependent on your JDBC driver support) (eg. `where date = {d '2008-12-31'}`)
- Java `public static final` constants `eg.Color.TABBY`

`in` and `between` may be used as follows:

```
select cat from DomesticCat cat where cat.name between 'A' and 'B'
```

```
select cat from DomesticCat cat where cat.name in ( 'Foo', 'Bar', 'Baz' )
```

and the negated forms may be written

```
select cat from DomesticCat cat where cat.name not between 'A' and 'B'
```

```
select cat from DomesticCat cat where cat.name not in ( 'Foo', 'Bar', 'Baz' )
```

Likewise, `is null` and `is not null` may be used to test for null values.

Booleans may be easily used in expressions by declaring HQL query substitutions in Hibernate configuration:

```
hibernate.query.substitutions true 1, false 0
```

This will replace the keywords `true` and `false` with the literals `1` and `0` in the translated SQL from this HQL:

```
select cat from Cat cat where cat.alive = true
```

You may test the size of a collection with the special property `size`, or the special `size()` function (HQL specific feature).

```
select cat from Cat cat where cat.kittens.size > 0
```

```
select cat from Cat cat where size(cat.kittens) > 0
```

For indexed collections, you may refer to the minimum and maximum indices using `minindex` and `maxindex` functions. Similarly, you may refer to the minimum and maximum elements of a collection of basic type using the `minelement` and `maxelement` functions. These are HQL specific features.

```
select cal from Calendar cal where maxelement(cal.holidays) > current date
```

```
select order from Order order where maxindex(order.items) > 100
```

```
select order from Order order where minelement(order.items) > 10000
```

The SQL functions `any, some, all, exists, in` are supported when passed the element or index set of a collection (`elements` and `indices` functions) or the result of a subquery (see below). While subqueries are supported by JP-QL, `elements` and `indices` are specific HQL features.

```
select mother from Cat as mother, Cat as kit
where kit in elements(foo.kittens)
```

```
select p from NameList list, Person p
where p.name = some elements(list.names)
```

```
select cat from Cat cat where exists elements(cat.kittens)
```

```
select cat from Player p where 3 > all elements(p.scores)
```

```
select cat from Show show where 'fizard' in indices(show.acts)
```

Note that these constructs - `size`, `elements`, `indices`, `minindex`, `maxindex`, `minelement`, `maxelement` - may only be used in the where clause in Hibernate.

JP-QL lets you access the key or the value of a map by using the `KEY()` and `VALUE()` operations (even access the Entry object using `ENTRY()`)

```
SELECT i.name, VALUE(p) FROM Item i JOIN i.photos p WHERE KEY(p) LIKE ‘%egret’
```

In HQL, elements of indexed collections (arrays, lists, maps) may be referred to by index (in a where clause only):

```
select order from Order order where order.items[0].id = 1234
```

```
select person from Person person, Calendar calendar
```

```
where calendar.holidays['national day'] = person.birthDay
    and person.nationality.calendar = calendar
```

```
select item from Item item, Order order
where order.items[ order.deliveredItemIndices[0] ] = item and order.id = 11
```

```
select item from Item item, Order order
where order.items[ maxindex(order.items) ] = item and order.id = 11
```

The expression inside `[]` may even be an arithmetic expression.

```
select item from Item item, Order order
where order.items[ size(order.items) - 1 ] = item
```

HQL also provides the built-in `index()` function, for elements of a one-to-many association or collection of values.

```
select item, index(item) from Order order
    join order.items item
where index(item) < 5
```

Scalar SQL functions supported by the underlying database may be used

```
select cat from DomesticCat cat where upper(cat.name) like 'FRI%'
```

If you are not yet convinced by all this, think how much longer and less readable the following query would be in SQL:

```
select cust
from Product prod,
    Store store
    inner join store.customers cust
where prod.name = 'widget'
    and store.location.name in ( 'Melbourne', 'Sydney' )
    and prod = all elements(cust.currentOrder.lineItems)
```

Hint: something like

```
SELECT cust.name, cust.address, cust.phone, cust.id, cust.current_order
FROM customers cust,
```

```
    stores store,
    locations loc,
    store_customers sc,
    product prod
WHERE prod.name = 'widget'
    AND store.loc_id = loc.id
    AND loc.name IN ( 'Melbourne', 'Sydney' )
    AND sc.store_id = store.id
    AND sc.cust_id = cust.id
    AND prod.id = ALL(
        SELECT item.prod_id
        FROM line_items item, orders o
        WHERE item.order_id = o.id
            AND cust.current_order = o.id
    )
```

8.9. The order by clause

The list returned by a query may be ordered by any property of a returned class or components:

```
select cat from DomesticCat cat
order by cat.name asc, cat.weight desc, cat.birthdate
```

The optional `asc` or `desc` indicate ascending or descending order respectively.

8.10. The group by clause

A query that returns aggregate values may be grouped by any property of a returned class or components:

```
select cat.color, sum(cat.weight), count(cat)
from Cat cat
group by cat.color
```

```
select foo.id, avg(name), max(name)
from Foo foo join foo.names name
group by foo.id
```

A `having` clause is also allowed.

```
select cat.color, sum(cat.weight), count(cat)
from Cat cat
group by cat.color
having cat.color in (eg.Color.TABBY, eg.Color.BLACK)
```

SQL functions and aggregate functions are allowed in the `having` and `order by` clauses, if supported by the underlying database (eg. not in MySQL).

```
select cat
from Cat cat
    join cat.kittens kitten
group by cat
having avg(kitten.weight) > 100
order by count(kitten) asc, sum(kitten.weight) desc
```

Note that neither the `group by` clause nor the `order by` clause may contain arithmetic expressions.

8.11. Subqueries

For databases that support subselects, JP-QL supports subqueries within queries. A subquery must be surrounded by parentheses (often by an SQL aggregate function call). Even correlated subqueries (subqueries that refer to an alias in the outer query) are allowed.

```
select fatcat from Cat as fatcat
where fatcat.weight > (
    select avg(cat.weight) from DomesticCat cat
)
```

```
select cat from DomesticCat as cat
where cat.name = some (
    select name.nickName from Name as name
)
```

```
select cat from Cat as cat
where not exists (
    from Cat as mate where mate.mate = cat
)
```

```
select cat from DomesticCat as cat
where cat.name not in (
    select name.nickName from Name as name
)
```

For subqueries with more than one expression in the select list, you can use a tuple constructor:

```
select cat from Cat as cat
where not ( cat.name, cat.color ) in (
```

```
    select cat.name, cat.color from DomesticCat cat
)
```

Note that on some databases (but not Oracle or HSQLDB), you can use tuple constructors in other contexts, for example when querying components or composite user types:

```
select cat from Person where name = ('Gavin', 'A', 'King')
```

Which is equivalent to the more verbose:

```
select cat from Person where name.first = 'Gavin' and name.initial = 'A' and name.last = 'King')
```

There are two good reasons you might not want to do this kind of thing: first, it is not completely portable between database platforms; second, the query is now dependent upon the ordering of properties in the mapping document.

8.12. JP-QL examples

Hibernate queries can be quite powerful and complex. In fact, the power of the query language is one of Hibernate's main selling points (and now JP-QL). Here are some example queries very similar to queries that I used on a recent project. Note that most queries you will write are much simpler than these!

The following query returns the order id, number of items and total value of the order for all unpaid orders for a particular customer and given minimum total value, ordering the results by total value. In determining the prices, it uses the current catalog. The resulting SQL query, against the `ORDER`, `ORDER_LINE`, `PRODUCT`, `CATALOG` and `PRICE` tables has four inner joins and an (uncorrelated) subselect.

```
select order.id, sum(price.amount), count(item)
from Order as order
    join order.lineItems as item
    join item.product as product,
    Catalog as catalog
    join catalog.prices as price
where order.paid = false
    and order.customer = :customer
    and price.product = product
    and catalog.effectiveDate < sysdate
    and catalog.effectiveDate >= all (
        select cat.effectiveDate
        from Catalog as cat
        where cat.effectiveDate < sysdate
    )
group by order
having sum(price.amount) > :minAmount
```

```
order by sum(price.amount) desc
```

What a monster! Actually, in real life, I'm not very keen on subqueries, so my query was really more like this:

```
select order.id, sum(price.amount), count(item)
from Order as order
    join order.lineItems as item
    join item.product as product,
    Catalog as catalog
    join catalog.prices as price
where order.paid = false
    and order.customer = :customer
    and price.product = product
    and catalog = :currentCatalog
group by order
having sum(price.amount) > :minAmount
order by sum(price.amount) desc
```

The next query counts the number of payments in each status, excluding all payments in the `AWAITING_APPROVAL` status where the most recent status change was made by the current user. It translates to an SQL query with two inner joins and a correlated subselect against the `PAYMENT`, `PAYMENT_STATUS` and `PAYMENT_STATUS_CHANGE` tables.

```
select count(payment), status.name
from Payment as payment
    join payment.currentStatus as status
    join payment.statusChanges as statusChange
where payment.status.name <> PaymentStatus.AWAITING_APPROVAL
    or (
        statusChange.timeStamp = (
            select max(change.timeStamp)
            from PaymentStatusChange change
            where change.payment = payment
        )
        and statusChange.user <> :currentUser
    )
group by status.name, status.sortOrder
order by status.sortOrder
```

If I would have mapped the `statusChanges` collection as a list, instead of a set, the query would have been much simpler to write.

```
select count(payment), status.name
from Payment as payment
    join payment.currentStatus as status
where payment.status.name <> PaymentStatus.AWAITING_APPROVAL
    or payment.statusChanges[ maxIndex(payment.statusChanges) ].user <> :currentUser
group by status.name, status.sortOrder
```

```
order by status.sortOrder
```

However the query would have been HQL specific.

The next query uses the MS SQL Server `isNull()` function to return all the accounts and unpaid payments for the organization to which the current user belongs. It translates to an SQL query with three inner joins, an outer join and a subselect against the `ACCOUNT`, `PAYMENT`, `PAYMENT_STATUS`, `ACCOUNT_TYPE`, `ORGANIZATION` and `ORG_USER` tables.

```
select account, payment
from Account as account
    join account.holder.users as user
    left outer join account.payments as payment
where :currentUser = user
    and PaymentStatus.UNPAID = isNull(payment.currentStatus.name, PaymentStatus.UNPAID)
order by account.type.sortOrder, account.accountNumber, payment.dueDate
```

8.13. Bulk UPDATE & DELETE Statements

Hibernate now supports UPDATE and DELETE statements in HQL/JP-QL. See *Section 7.1, "Bulk update/delete"* for details.

8.14. Tips & Tricks

To order a result by the size of a collection, use the following query:

```
select usr.id, usr.name
from User as usr
    left join usr.messages as msg
group by usr.id, usr.name
order by count(msg)
```

If your database supports subselects, you can place a condition upon selection size in the where clause of your query:

```
from User usr where size(usr.messages) >= 1
```

If your database doesn't support subselects, use the following query:

```
select usr.id, usr.name
from User usr.name
    join usr.messages msg
group by usr.id, usr.name
having count(msg) >= 1
```

As this solution can't return a `User` with zero messages because of the inner join, the following form is also useful:

```
select usr.id, usr.name
from User as usr
    left join usr.messages as msg
group by usr.id, usr.name
having count(msg) = 0
```

Criteria Queries

Criteria queries are a programmatic, type-safe way to express a query. They are type-safe in terms of using interfaces and classes to represent various structural parts of a query such as the query itself, or the select clause, or an order-by, etc. They can also be type-safe in terms of referencing attributes as we will see in a bit. Users of the older Hibernate `org.hibernate.Criteria` query API will recognize the general approach, though we believe the JPA API to be superior as it represents a clean look at the lessons learned from that API.

Criteria queries are essentially an object graph, where each part of the graph represents an increasing (as we navigate down this graph) more atomic part of query. The first step in performing a criteria query is building this graph. The `javax.persistence.criteria.CriteriaBuilder` interface is the first thing with which you need to become acquainted to begin using criteria queries. Its role is that of a factory for all the individual pieces of the criteria. You obtain a `javax.persistence.criteria.CriteriaBuilder` instance by calling the `getCriteriaBuilder` method of the `javax.persistence.EntityManagerFactory`

```
CriteriaBuilder builder = entityManagerFactory.getCriteriaBuilder();
```

The next step is to obtain a `javax.persistence.criteria.CriteriaQuery`. You do this by one of the 3 methods on `javax.persistence.criteria.CriteriaBuilder` for this purpose.

```
CriteriaQuery<T> createQuery(Class<T>)
```

```
CriteriaQuery<Tuple> createTupleQuery()
```

```
CriteriaQuery<Object> createQuery()
```

Each serves a different purpose depending on the expected type of the query results.

Note

Chapter 6 Criteria API of the [*JPA 2 Specification*] already contains a decent amount of reference material pertaining to the various parts of a criteria query. So rather than duplicate all that content here, lets instead look at some of the more widely anticipated usages of the API.

9.1. Typed criteria queries

```
CriteriaQuery<T> createQuery(Class<T>)
```

The type of the criteria query (aka the <T>) indicates the expected types in the query result. This might be an entity, an Integer, or any other object.

9.1.1. Selecting an entity

This the most used form of query in Hibernate Query Language (HQL) and Hibernate Criteria Queries. You have an entity and you want to select one or more of that entity based on some condition.

Example 9.1. Selecting the root entity

```
CriteriaQuery<Person> criteria = builder.createQuery( Person.class );
Root<Person> personRoot = criteria.from( Person.class );

criteria.select( personRoot );

criteria.where( builder.equal( personRoot.get( Person_.eyeColor ), "brown" ) );
List<Person> people = em.createQuery( criteria ).getResultList();
for ( Person person : people ) { ... }
```

1. We use the form *createQuery(Person.class)* here because the expected returns are in fact Person entities as we see when we begin processing the results.
2. *personCriteria.select(personRoot)* here is completely unneeded in this specific case because of the fact that *personRoot* will be the implied selection since we have only a single root. It was done here only for completeness of an example
3. *Person_.eyeColor* is an example of the static form of metamodel reference. We will use that form exclusively in this chapter. See *Section 4.1, "Static metamodel"* for details.

9.1.2. Selecting a value

The simplest form of selecting a value is selecting a particular attribute from an entity. But this might also be an aggregation, a mathematical operation, etc.

Example 9.2. Selecting an attribute

```
CriteriaQuery<Integer> criteria = builder.createQuery( Integer.class );
Root<Person> personRoot = criteria.from( Person.class );

criteria.select( personRoot.get( Person_.age ) );
criteria.where( builder.equal( personRoot.get( Person_.eyeColor ), "brown" ) );

List<Integer> ages = em.createQuery( criteria ).getResultList();
```

```
for ( Integer age : ages ) { ... }
```

1. Notice again the typing of the query based on the anticipated result type(s). Here we are specifying `java.lang.Integer` as the type of the *Person#age* attribute is `java.lang.Integer`.
2. We need to bind the fact that we are interested in the age associated with the *personRoot*. We might have multiple references to the Person entity in the query so we need to identify (aka qualify) which *Person#age* we mean.

Example 9.3. Selecting an expression

```
CriteriaQuery<Integer> criteria = builder.createQuery( Integer.class );
Root<Person> personRoot = criteria.from( Person.class );

criteria.select( builder.max( personRoot.get( Person_.age ) ) );                (1)
criteria.where( builder.equal( personRoot.get( Person_.eyeColor ), "brown" ) );
Integer maxAge = em.createQuery( criteria ).getSingleResult();
```

1. Here we see `javax.persistence.criteria.CriteriaBuilder` used to obtain a *MAX* expression. These expression building methods return `javax.persistence.criteria.Expression` instances typed according to various rules. The rule for a *MAX* expression is that the expression type is the same as that of the underlying attribute.

9.1.3. Selecting multiple values

There are actually a few different ways to select multiple values using criteria queries. We will explore 2 options here, but an alternative recommended approach is to use tuples as described in *Section 9.2, "Tuple criteria queries"*

Example 9.4. Selecting an array

```
CriteriaQuery<Object[]> criteria = builder.createQuery( Object[].class );      (1)
Root<Person> personRoot = criteria.from( Person.class );
Path<Long> idPath = personRoot.get( Person_.id );
Path<Integer> agePath = personRoot.get( Person_.age );

criteria.select( builder.array( idPath, agePath ) );                            (2)
criteria.where( builder.equal( personRoot.get( Person_.eyeColor ), "brown" ) );

List<Object[]> valueArray = em.createQuery( criteria ).getResultList();        (1)
for ( Object[] values : valueArray ) {
    final Long id = (Long) values[0];
    final Integer age = (Integer) values[1];
    ...
}
```

1. Technically this is classified as a typed query, but as you can see in handling the results that is sort of misleading. Anyway, the expected result type here is an array.
2. Here we see the use of the `array` method of the `javax.persistence.criteria.CriteriaBuilder` which explicitly combines individual selections into a `javax.persistence.criteria.CompoundSelection`.

Example 9.5. Selecting an array (2)

```
CriteriaQuery<Object[]> criteria = builder.createQuery( Object[].class );          (1)
Root<Person> personRoot = criteria.from( Person.class );
Path<Long> idPath = personRoot.get( Person_.id );
Path<Integer> agePath = personRoot.get( Person_.age );

criteria.multiselect( idPath, agePath );                                           (2)
criteria.where( builder.equal( personRoot.get( Person_.eyeColor ), "brown" ) );

List<Object[]> valueArray = em.createQuery( criteria ).getResultList();            (1)
for ( Object[] values : valueArray ) {
    final Long id = (Long) values[0];
    final Integer age = (Integer) values[1];
    ...
}
```

1. Just as we saw in *Example 9.4, "Selecting an array"* we have a "typed" criteria query returning an Object array.
2. This actually functions exactly the same as what we saw in *Example 9.4, "Selecting an array"*. The `multiselect` method behaves slightly differently based on the type given when the criteria query was first built, but in this case it says to select and return an *Object[]*.

9.1.4. Selecting a wrapper

Another alternative to *Section 9.1.3, "Selecting multiple values"* is to instead select an object that will "wrap" the multiple values. Going back to the example query there, rather than returning an array of *[Person#id, Person#age]* instead declare a class that holds these values and instead return that.

Example 9.6. Selecting an wrapper

```
public class PersonWrapper {                                                       (1)
    private final Long id;
    private final Integer age;

    public PersonWrapper(Long id, Integer age) {                                   (1)
        this.id = id;
        this.age = age;
    }
    ...
}
...
CriteriaQuery<PersonWrapper> criteria = builder.createQuery( PersonWrapper.class ); (2)
```

```
Root<Person> personRoot = criteria.from( Person.class );
criteria.select(                                                    (3)
    builder.construct(                                              (3)
        PersonWrapper.class,
        personRoot.get( Person_.id ),
        personRoot.get( Person_.age )
    )
);
criteria.where( builder.equal( personRoot.get( Person_.eyeColor ), "brown" ) );
List<PersonWrapper> people = em.createQuery( criteria ).getResultList();   (2)
for ( PersonWrapper person : people ) { ... }
```

(1) First we see the simple definition of the wrapper object we will be using to wrap our result values. Specifically notice the constructor and its argument types.

(2) Since we will be returning *PersonWrapper* objects, we use *PersonWrapper* as the type of our criteria query.

(3) Here we see another new `javax.persistence.criteria.CriteriaBuilder` method, `construct`, which is used to builder a wrapper expression. Basically for every row in the result we are saying we would like a *PersonWrapper* instantiated by the matching constructor. This wrapper expression is then passed as the select.

9.2. Tuple criteria queries

A better approach to *Section 9.1.3, "Selecting multiple values"* is to either use a wrapper (which we just saw in *Section 9.1.4, "Selecting a wrapper"*) or using the `javax.persistence.Tuple` contract.

Example 9.7. Selecting a tuple

```
CriteriaQuery<Tuple> criteria = builder.createTupleQuery();          (1)
Root<Person> personRoot = criteria.from( Person.class );
Path<Long> idPath = personRoot.get( Person_.id );
Path<Integer> agePath = personRoot.get( Person_.age );
criteria.multiselect( idPath, agePath );                             (2)
criteria.where( builder.equal( personRoot.get( Person_.eyeColor ), "brown" ) );
List<Tuple> tuples = em.createQuery( criteria ).getResultList();     (1)
for ( Tuple tuple : valueArray ) {
    assert tuple.get( 0 ) == tuple.get( idPath );                    (3)
    assert tuple.get( 1 ) == tuple.get( agePath );                   (3)
    ...
}
```

(1) Here we see the use of a new `javax.persistence.criteria.CriteriaBuilder` `javax.persistence.criteria.CriteriaQuery` building method, `createTupleQuery`. This is exactly equivalent to calling *builder.createQuery(Tuple.class)*. It signifies that we want to access the results through the `javax.persistence.Tuple` contract.

2. Again we see the use of the `multiselect` method, just like in *Example 9.5, "Selecting an array (2)"*. The difference here is that the type of the `javax.persistence.criteria.CriteriaQuery` was defined as `javax.persistence.Tuple` so the compound selections in this case are interpreted to be the tuple elements.
3. Here we see `javax.persistence.Tuple` allowing different types of access to the results, which we will expand on next.

9.2.1. Accessing tuple elements

The `javax.persistence.Tuple` contract provides 3 basic forms of access to the underlying elements:

typed

```
<X> X get(TupleElement<X> tupleElement)
```

This allows typed access to the underlying tuple elements. We see this in *Example 9.7, "Selecting a tuple"* in the *tuple.get(idPath)* and *tuple.get(agePath)* calls. Just about everything is a `javax.persistence.TupleElement`.

positional

```
Object get(int i)
```

```
<X> X get(int i, Class<X> type)
```

Very similar to what we saw in *Example 9.4, "Selecting an array"* and *Example 9.5, "Selecting an array (2)"* in terms of positional access. Only the second form here provides typing, because the user explicitly provides the typing on access. We see this in *Example 9.7, "Selecting a tuple"* in the *tuple.get(0)* and *tuple.get(1)* calls.

aliased

```
Object get(String alias)
```

```
<X> X get(String alias, Class<X> type)
```

Again, only the second form here provides typing, because the user explicitly provides the typing on access. We have not seen an example of using this, but its trivial. We would

simply, for example, have applies an alias to either of the paths like *idPath.alias("id")* and/or *agePath.alias("age")* and we could have accessed the individual tuple elements by those specified aliases.

9.3. FROM clause

> A CriteriaQuery object defines a query over one or more entity, embeddable, or basic abstract schema types. The root objects of the query are entities, from which the other types are reached by navigation.
>
> —[*JPA 2 Specification*, section 6.5.2 Query Roots, pg 262]

Note

All the individual parts of the FROM clause (roots, joins, paths) implement the `javax.persistence.criteria.From` interface.

9.3.1. Roots

Roots define the basis from which all joins, paths and attributes are available in the query. In a criteria query, a root is always an entity. Roots are defined and added to the criteria by the overloaded `from` methods on `javax.persistence.criteria.CriteriaQuery`:

```
<X> Root<X> from(Class<X>)
```

```
<X> Root<X> from(EntityType<X>)
```

Example 9.8. Adding a root

```
CriteriaQuery<Person> personCriteria = builder.createQuery( Person.class );
// create and add the root
person.from( Person.class );
...
```

Criteria queries may define multiple roots, the effect of which is to create a *cartesian product* [http://en.wikipedia.org/wiki/Cartesian_product] between the newly added root and the others. Here is an example matching all single men and all single women:

```
CriteriaQuery query = builder.createQuery();
Root<Person> men = query.from( Person.class );
Root<Person> women = query.from( Person.class );
Predicate menRestriction = builder.and(
```

```
    builder.equal( men.get( Person_.gender ), Gender.MALE ),
    builder.equal( men.get( Person_.relationshipStatus ), RelationshipStatus.SINGLE )
);
Predicate womenRestriction = builder.and(
    builder.equal( women.get( Person_.gender ), Gender.FEMALE ),
    builder.equal( women.get( Person_.relationshipStatus ), RelationshipStatus.SINGLE )
);
query.where( builder.and( menRestriction, womenRestriction ) );
```

9.3.2. Joins

Joins allow navigation from other `javax.persistence.criteria.From` to either association or embedded attributes. Joins are created by the numerous overloaded `join` methods of the `javax.persistence.criteria.From` interface:

Example 9.9. Example with Embedded and ManyToOne

```
CriteriaQuery<Person> personCriteria = builder.createQuery( Person.class );
Root<Person> personRoot = person.from( Person.class );
// Person.address is an embedded attribute
Join<Person,Address> personAddress = personRoot.join( Person_.address );
// Address.country is a ManyToOne
Join<Address,Country> addressCountry = personAddress.join( Address_.country );
...
```

Example 9.10. Example with Collections

```
CriteriaQuery<Person> personCriteria = builder.createQuery( Person.class );
Root<Person> personRoot = person.from( Person.class );
Join<Person,Order> orders = personRoot.join( Person_.orders );
Join<Order,LineItem> orderLines = orders.join( Order_.lineItems );
...
```

9.3.3. Fetches

Just like in HQL and EJB-QL, we can specify that associated data be fetched along with the owner. Fetches are created by the numerous overloaded `fetch` methods of the `javax.persistence.criteria.From` interface:

Example 9.11. Example with Embedded and ManyToOne

```
CriteriaQuery<Person> personCriteria = builder.createQuery( Person.class );
Root<Person> personRoot = person.from( Person.class );
// Person.address is an embedded attribute
Join<Person,Address> personAddress = personRoot.fetch( Person_.address );
// Address.country is a ManyToOne
Join<Address,Country> addressCountry = personAddress.fetch( Address_.country );
```

```
...
```

Note

Technically speaking, embedded attributes are always fetched with their owner. However in order to define the fetching of *Address#country* we needed a `javax.persistence.criteria.Fetch` for its parent path.

Example 9.12. Example with Collections

```
CriteriaQuery<Person> personCriteria = builder.createQuery( Person.class );
Root<Person> personRoot = person.from( Person.class );
Join<Person,Order> orders = personRoot.fetch( Person_.orders );
Join<Order,LineItem> orderLines = orders.fetch( Order_.lineItems );
...
```

9.4. Path expressions

Note

Roots, joins and fetches are themselves paths as well.

9.5. Using parameters

Example 9.13. Using parameters

```
CriteriaQuery<Person> criteria = build.createQuery( Person.class );
Root<Person> personRoot = criteria.from( Person.class );
criteria.select( personRoot );

ParameterExpression<String> eyeColorParam = builder.parameter( String.class ); (1)

criteria.where( builder.equal( personRoot.get( Person_.eyeColor ), eyeColorParam ) ); (2)
TypedQuery<Person> query = em.createQuery( criteria );

query.setParameter( eyeColorParam, "brown" ); (3)
List<Person> people = query.getResultList();
```

1. Use the `parameter` method of `javax.persistence.criteria.CriteriaBuilder` to obtain a parameter reference.
2. Use the parameter reference in the criteria query.
3. Use the parameter reference to bind the parameter value to the `javax.persistence.TypedQuery`

Native query

You may also express queries in the native SQL dialect of your database. This is useful if you want to utilize database specific features such as query hints or the CONNECT BY option in Oracle. It also provides a clean migration path from a direct SQL/JDBC based application to Hibernate. Note that Hibernate allows you to specify handwritten SQL (including stored procedures) for all create, update, delete, and load operations (please refer to the reference guide for more information.)

10.1. Expressing the resultset

To use a SQL query, you need to describe the SQL resultset, this description will help the `EntityManager` to map your columns onto entity properties. This is done using the `@SqlResultSetMapping` annotation. Each `@SqlResultSetMapping` has a name which is used when creating a SQL query on `EntityManager`.

```
@SqlResultSetMapping(name="GetNightAndArea", entities={
    @EntityResult(name="org.hibernate.test.annotations.query.Night", fields = {
        @FieldResult(name="id", column="nid"),
        @FieldResult(name="duration", column="night_duration"),
        @FieldResult(name="date", column="night_date"),
        @FieldResult(name="area", column="area_id")
    }),
    @EntityResult(name="org.hibernate.test.annotations.query.Area", fields = {
        @FieldResult(name="id", column="aid"),
        @FieldResult(name="name", column="name")
    })
    }
)

//or
@SqlResultSetMapping(name="defaultSpaceShip", entities=@EntityResult(name="org.hibernate.test.annotations.query.Sp
```

You can also define scalar results and even mix entity results and scalar results

```
@SqlResultSetMapping(name="ScalarAndEntities",
    entities={
        @EntityResult(name="org.hibernate.test.annotations.query.Night", fields = {
            @FieldResult(name="id", column="nid"),
            @FieldResult(name="duration", column="night_duration"),
            @FieldResult(name="date", column="night_date"),
            @FieldResult(name="area", column="area_id")
        }),
        @EntityResult(name="org.hibernate.test.annotations.query.Area", fields = {
            @FieldResult(name="id", column="aid"),
            @FieldResult(name="name", column="name")
        })
    },
    columns={
        @ColumnResult(name="durationInSec")
    }
```

```
)
```

The SQL query will then have to return a column alias `durationInSec`.

Please refer to the Hibernate Annotations reference guide for more information about `@SqlResultSetMapping`.

10.2. Using native SQL Queries

TODO: This sounds like a dupe...

Now that the result set is described, we are capable of executing the native SQL query. `EntityManager` provides all the needed APIs. The first method is to use a SQL resultset name to do the binding, the second one uses the entity default mapping (the column returned has to have the same names as the one used in the mapping). A third one (not yet supported by Hibernate entity manager), returns pure scalar results.

```
String sqlQuery = "select night.id nid, night.night_duration, night.night_date, area.id aid, "
    + "night.area_id, area.name from Night night, Area area where night.area_id = area.id "
    + "and night.night_duration >= ?";
Query q = entityManager.createNativeQuery(sqlQuery, "GetNightAndArea");
q.setParameter( 1, expectedDuration );
q.getResultList();
```

This native query returns nights and area based on the `GetNightAndArea` result set.

```
String sqlQuery = "select * from tbl_spaceship where owner = ?";
Query q = entityManager.createNativeQuery(sqlQuery, SpaceShip.class);
q.setParameter( 1, "Han" );
q.getResultList();
```

The second version is useful when your SQL query returns one entity reusing the same columns as the ones mapped in metadata.

10.3. Named queries

Native named queries share the same calling API than JP-QL named queries. Your code doesn't need to know the difference between the two. This is very useful for migration from SQL to JP-QL:

```
Query q = entityManager.createNamedQuery("getSeasonByNativeQuery");
q.setParameter( 1, name );
Season season = (Season) q.getSingleResult();
```

References

[JPA 2 Specification] *JSR 317: Java™ Persistence API, Version 2.0*. Java Persistence 2.0 Expert Group. . Copyright © 2009 SUN MICROSYSTEMS, INC.. <jsr-317-feedback@sun.com> *JSR 317 JCP Page* [http://jcp.org/en/jsr/detail?id=317].