◆ ◆ ◆ CHAPTER 7

Using the Java Persistence API

Oracle GlassFish Server support for the Java Persistence API includes all required features described in the Java Persistence Specification, also known as JSR 317 (`http://jcp.org/en/jsr/detail?id=317`). The Java Persistence API can be used with non-EJB components outside the EJB container.

The Java Persistence API provides an object/relational mapping facility to Java developers for managing relational data in Java applications. For basic information about the Java Persistence API, see Part VI, "Persistence," in *The Java EE 6 Tutorial*.

This chapter contains GlassFish Server specific information on using the Java Persistence API in the following topics:

Note – The default persistence provider in the GlassFish Server is based on the EclipseLink Java Persistence API implementation. All configuration options in EclipseLink are available to applications that use the GlassFish Server's default persistence provider.

Note – The Web Profile of the GlassFish Server supports the EJB 3.1 Lite specification, which allows enterprise beans within web applications, among other features. The full GlassFish Server supports the entire EJB 3.1 specification. For details, see JSR 318 (`http://jcp.org/en/jsr/detail?id=318`).

Specifying the Database

The GlassFish Server uses the bundled Java DB (Derby) database by default, named `jdbc/__default`. If the `transaction-type` element is omitted or specified as `JTA` and both the `jta-data-source` and `non-jta-data-source` elements are omitted in the `persistence.xml` file, Java DB is used as a JTA data source. If `transaction-type` is specified as `RESOURCE_LOCAL` and both `jta-data-source` and `non-jta-data-source` are omitted, Java DB is used as a non-JTA data source.

To use a non-default database, either specify a value for the `jta-data-source` element, or set the `transaction-type` element to `RESOURCE_LOCAL` and specify a value for the `non-jta-data-source` element.

If you are using the default persistence provider, the provider attempts to automatically detect the database type based on the connection metadata. This database type is used to issue SQL statements specific to the detected database type's dialect. You can specify the optional `eclipselink.target-database` property to guarantee that the database type is correct. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
    <persistence xmlns="http://java.sun.com/xml/ns/persistence">
        <persistence-unit name ="em1">
            <jta-data-source>jdbc/MyDB2DB</jta-data-source>
            <properties>
                <property name="eclipselink.target-database"
                    value="DB2"/>
            </properties>
        </persistence-unit>
    </persistence>
```

The following `eclipselink.target-database` property values are allowed. Supported platforms have been tested with the GlassFish Server and are found to be Java EE compatible.

```
//Supported platforms
JavaDB
Derby
Oracle
MySQL4
//Others available
SQLServer
DB2
Sybase
```

```
PostgreSQL
Informix
TimesTen
Attunity
HSQL
SQLAnyWhere
DBase
DB2Mainframe
Cloudscape
PointBase
```

For more information about the `eclipselink.target-database` property, see Using EclipseLink JPA Extensions for Session, Target Database and Target Application Server.

To use the Java Persistence API outside the EJB container (in Java SE mode), do not specify the `jta-data-source` or `non-jta-data-source` elements. Instead, specify the `provider` element and any additional properties required by the JDBC driver or the database. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
    <persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">
        <persistence-unit name ="em2">
            <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
            <class>ejb3.war.servlet.JpaBean</class>
            <properties>
                <property name="eclipselink.target-database"
                    value="Derby"/>
                <!-- JDBC connection properties -->
                <property name="eclipselink.jdbc.driver" value="org.apache.derby.jdbc.ClientDriver"/>
                <property name="eclipselink.jdbc.url"
value="jdbc:derby://localhost:1527/testdb;retrieveMessagesFromServerOnGetMessage=true;create=true;"/>
                <property name="eclipselink.jdbc.user" value="APP"/>
                <property name="eclipselink.jdbc.password" value="APP"/>
            </properties>
        </persistence-unit>
    </persistence>
```

For more information about `eclipselink` properties, see "Additional Database Properties" on page 20.

For a list of the JDBC drivers currently supported by the GlassFish Server, see the *GlassFish Server Open Source Edition 3.1 Release Notes*. For configurations of supported and other drivers, see "Configuration Specifics for JDBC Drivers" in *GlassFish Server Open Source Edition 3.1 Administration Guide*.

To change the persistence provider, see "Changing the Persistence Provider" on page 24.

Additional Database Properties

If you are using the default persistence provider, you can specify in the `persistence.xml` file the database properties listed at How to Use EclipseLink JPA Extensions for JDBC Connection Communication.

For schema generation properties, see "Generation Options" on page 22. For query hints, see "Query Hints" on page 24.

Configuring the Cache

If you are using the default persistence provider, you can configure whether caching occurs, the type of caching, the size of the cache, and whether client sessions share the cache. Caching properties for the default persistence provider are described in detail at Using EclipseLink JPA Extensions for Entity Caching.

Setting the Logging Level

One of the default persistence provider's properties that you can set in the `persistence.xml` file is `eclipselink.logging.level`. For example, setting the logging level to `FINE` or higher logs all SQL statements. For details about this property, see Using EclipseLink JPA Extensions for Logging.

You can also set the EclipseLink logging level globally in the GlassFish Server by setting a JVM option using the `asadmin create-jvm-options` command. For example:

```
asadmin create-jvm-options -Declipselink.logging.level=FINE
```

Setting the logging level to `OFF` disables EclipseLink logging. A logging level set in the `persistence.xml` file takes precedence over the global logging level.

Using Lazy Loading

`OneToMany` and `ManyToMany` mappings are loaded lazily by default in compliance with the Java Persistence Specification. `OneToOne` and `ManyToOne` mappings are loaded eagerly by default.

For basic information about lazy loading, see What You May Need to Know About EclipseLink JPA Lazy Loading.

Primary Key Generation Defaults

In the descriptions of the `@GeneratedValue`, `@SequenceGenerator`, and `@TableGenerator` annotations in the Java Persistence Specification, certain defaults are noted as specific to the persistence provider. The default persistence provider's primary key generation defaults are listed here.

`@GeneratedValue` defaults are as follows:

- Using `strategy=AUTO` (or no `strategy`) creates a `@TableGenerator` named `SEQ_GEN` with default settings. Specifying a `generator` has no effect.
- Using `strategy=TABLE` without specifying a `generator` creates a `@TableGenerator` named `SEQ_GEN_TABLE` with default settings. Specifying a `generator` but no `@TableGenerator` creates and names a `@TableGenerator` with default settings.
- Using `strategy=IDENTITY` or `strategy=SEQUENCE` produces the same results, which are database-specific.
  - For Oracle databases, not specifying a `generator` creates a `@SequenceGenerator` named `SEQ_GEN_SEQUENCE` with default settings. Specifying a `generator` but no `@SequenceGenerator` creates and names a `@SequenceGenerator` with default settings.
  - For PostgreSQL databases, a `SERIAL` column named *entity-table_pk-column*`_SEQ` is created.
  - For MySQL databases, an `AUTO_INCREMENT` column is created.
  - For other supported databases, an `IDENTITY` column is created.

The `@SequenceGenerator` annotation has one default specific to the default provider. The default `sequenceName` is the specified `name`.

`@TableGenerator` defaults are as follows:

- The default `table` is `SEQUENCE`.
- The default `pkColumnName` is `SEQ_NAME`.
- The default `valueColumnName` is `SEQ_COUNT`.
- The default `pkColumnValue` is the specified `name`, or the default `name` if no `name` is specified.

Automatic Schema Generation

The automatic schema generation feature of the GlassFish Server defines database tables based on the fields or properties in entities and the relationships between the fields or properties. This insulates developers from many of the database related aspects of development, allowing them to focus on entity development. The resulting schema is usable as-is or can be given to a database administrator for tuning with respect to performance, security, and so on. This section covers the following topics:

Note – Automatic schema generation is supported on an all-or-none basis: it expects that no tables exist in the database before it is executed. It is not intended to be used as a tool to generate extra tables or constraints.

Deployment won't fail if all tables are not created, and undeployment won't fail if not all tables are dropped. Instead, an error is written to the server log. This is done to allow you to investigate the problem and fix it manually. You should not rely on the partially created database schema to be correct for running the application.

Annotations

The following annotations are used in automatic schema generation: `@AssociationOverride`, `@AssociationOverrides`, `@AttributeOverride`, `@AttributeOverrides`, `@Column`, `@DiscriminatorColumn`, `@DiscriminatorValue`, `@Embedded`, `@EmbeddedId`, `@GeneratedValue`, `@Id`, `@IdClass`, `@JoinColumn`, `@JoinColumns`, `@JoinTable`, `@Lob`, `@ManyToMany`, `@ManyToOne`, `@OneToMany`, `@OneToOne`, `@PrimaryKeyJoinColumn`, `@PrimaryKeyJoinColumns`, `@SecondaryTable`, `@SecondaryTables`, `@SequenceGenerator`, `@Table`, `@TableGenerator`, `@UniqueConstraint`, and `@Version`. For information about these annotations, see the Java Persistence Specification.

For `@Column` annotations, the `insertable` and `updatable` elements are not used in automatic schema generation.

For `@OneToMany` and `@ManyToOne` annotations, no `ForeignKeyConstraint` is created in the resulting DDL files.

Generation Options

Schema generation properties or `asadmin` command line options can control automatic schema generation by the following:

- Creating tables during deployment
- Dropping tables during undeployment
- Dropping and creating tables during redeployment
- Generating the DDL files

Note – Before using these options, make sure you have a properly configured database. See "Specifying the Database" on page 18.

Optional schema generation properties control the automatic creation of database tables. You can specify them in the `persistence.xml` file. For more information, see Using EclipseLink JPA Extensions for Schema Generation.

The following options of the `asadmin deploy` or `asadmin deploydir` command control the automatic creation of database tables at deployment.

TABLE 7–1 The `asadmin deploy` and `asadmin deploydir` Generation Options

Option	Default	Description
`--createtables`	none	If `true`, causes database tables to be created for entities that need them. No unique constraints are created. If `false`, does not create tables. If not specified, the value of the `eclipselink.ddl-generation` property in `persistence.xml` is used.
`--dropandcreatetables`	none	If `true`, and if tables were automatically created when this application was last deployed, tables from the earlier deployment are dropped and fresh ones are created. If `true`, and if tables were *not* automatically created when this application was last deployed, no attempt is made to drop any tables. If tables with the same names as those that would have been automatically created are found, the deployment proceeds, but a warning is thrown to indicate that tables could not be created. If `false`, the `eclipselink.ddl-generation` property setting in `persistence.xml` is overridden.

The following options of the `asadmin undeploy` command control the automatic removal of database tables at undeployment.

TABLE 7–2 The `asadmin undeploy` Generation Options

Option	Default	Description
`--droptables`	none	If `true`, causes database tables that were automatically created when the entities were last deployed to be dropped when the entities are undeployed. If `false`, does not drop tables. If not specified, tables are dropped only if the `eclipselink.ddl-generation` property setting in `persistence.xml` is `drop-and-create-tables`.

For more information about the `asadmin deploy`, `asadmin deploydir`, and `asadmin undeploy` commands, see the *GlassFish Server Open Source Edition 3.1 Reference Manual*.

When `asadmin` deployment options and `persistence.xml` options are both specified, the `asadmin` deployment options take precedence.

Query Hints

Query hints are additional, implementation-specific configuration settings. You can use hints in your queries in the following format:

```
setHint("hint-name", hint-value)
```

For example:

```
Customer customer = (Customer)entityMgr.
     createNamedQuery("findCustomerBySSN").
     setParameter("SSN", "123-12-1234").
     setHint("eclipselink.refresh", true).
     getSingleResult();
```

For more information about the query hints available with the default provider, see How to Use EclipseLink JPA Query Hints.

Changing the Persistence Provider

Note – The previous sections in this chapter apply only to the default persistence provider. If you change the provider for a module or application, the provider-specific database properties, query hints, and schema generation features described in this chapter do not apply.

You can change the persistence provider for an application in the manner described in the Java Persistence API Specification.

First, install the provider. Copy the provider JAR files to the *domain-dir*`/lib` directory, and restart the GlassFish Server. For more information about the *domain-dir*`/lib` directory, see "Using the Common Class Loader" on page . The new persistence provider is now available to all modules and applications deployed on servers that share the same configuration. However, the *default* provider remains the same.

In your persistence unit, specify the provider and any properties the provider requires in the `persistence.xml` file. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
    <persistence xmlns="http://java.sun.com/xml/ns/persistence">
        <persistence-unit name ="em3">
            <provider>com.company22.persistence.PersistenceProviderImpl</provider>
            <properties>
                <property name="company22.database.name" value="MyDB"/>
            </properties>
        </persistence-unit>
    </persistence>
```

To migrate from Oracle TopLink to EclipseLink, see Migrating from Oracle TopLink to EclipseLink (`http://wiki.eclipse.org/EclipseLink/Examples/MigratingFromOracleTopLink`).

Restrictions and Optimizations

This section discusses restrictions and performance optimizations that affect using the Java Persistence API.

Oracle Database Enhancements

EclipseLink features a number of enhancements for use with Oracle databases. These enhancements require classes from the Oracle JDBC driver JAR files to be visible to EclipseLink at runtime. If you place the JDBC driver JAR files in *domain-dir*`/lib`, the classes are not visible to GlassFish Server components, including EclipseLink.

If you are using an Oracle database, put JDBC driver JAR files in *domain-dir*`/lib/ext` instead. This ensures that the JDBC driver classes are visible to EclipseLink.

If you do not want to take advantage of Oracle-specific extensions from EclipseLink or you cannot put JDBC driver JAR files in *domain-dir*`/lib/ext`, set the `eclipselink.target-database` property to the value `org.eclipse.persistence.platform.database.OraclePlatform`. For more information about the `eclipselink.target-database` property, see "Specifying the Database" on page 18.

Extended Persistence Context

The Java Persistence API specification does not specify how the container and persistence provider should work together to serialize an extended persistence context. This also prevents successful serialization of a reference to an extended persistence context in a stateful session bean.

Even in a single-instance environment, if a stateful session bean is passivated, its extended persistence context could be lost when the stateful session bean is activated.

Therefore, in GlassFish Server, a stateful session bean with an extended persistence context is never passivated and cannot be failed over.

Using @OrderBy with a Shared Session Cache

Setting `@OrderBy` on a `ManyToMany` or `OneToMany` relationship field in which a `List` represents the Many side doesn't work if the session cache is shared. Use one of the following workarounds:

- Have the application maintain the order so the `List` is cached properly.
- Refresh the session cache using `EntityManager.refresh()` if you don't want to maintain the order during creation or modification of the `List`.
- Disable session cache sharing in `persistence.xml` as follows:

```
<property name="eclipselink.cache.shared.default" value="false"/>
```

Using BLOB or CLOB Types with the Inet Oraxo JDBC Driver

To use BLOB or CLOB data types larger than 4 KB for persistence using the Inet Oraxo JDBC Driver for Oracle Databases, you must set the database's `streamstolob` property value to `true`.

Database Case Sensitivity

Mapping references to column or table names must be in accordance with the expected column or table name case, and ensuring this is the programmer's responsibility. If column or table names are not explicitly specified for a field or entity, the GlassFish Server uses upper case column names by default, so any mapping references to the column or table names must be in upper case. If column or table names are explicitly specified, the case of all mapping references to the column or table names must be in accordance with the case used in the specified names.

The following are examples of how case sensitivity affects mapping elements that refer to columns or tables. Programmers must keep case sensitivity in mind when writing these mappings.

Unique Constraints

If column names are not explicitly specified on a field, unique constraints and foreign key mappings must be specified using uppercase references. For example:

```
@Table(name="Department", uniqueConstraints={ @UniqueConstraint ( columnNames= { "DEPTNAME" } ) } )
```

The other way to handle this is by specifying explicit column names for each field with the required case. For example:

```
@Table(name="Department", uniqueConstraints={ @UniqueConstraint ( columnNames= { "deptName" } ) } )
public class Department{ @Column(name="deptName") private String deptName; }
```

Otherwise, the `ALTER TABLE` statement generated by the GlassFish Server uses the incorrect case, and the creation of the unique constraint fails.

Foreign Key Mapping

Use `@OneToMany(mappedBy="COMPANY")` or specify an explicit column name for the `Company` field on the `Many` side of the relationship.

SQL Result Set Mapping

Use the following elements:

```
<sql-result-set-mapping name="SRSMName" >
   <entity-result entity-class="entities.someEntity" />
   <column-result name="UPPERCASECOLUMNNAME" />
</sql-result-set-mapping>
```

Or specify an explicit column name for the `upperCaseColumnName` field.

Named Native Queries and JDBC Queries

Column or table names specified in SQL queries must be in accordance with the expected case. For example, MySQL requires column names in the `SELECT` clause of JDBC queries to be uppercase, while PostgreSQL and Sybase require table names to be uppercase in all JDBC queries.

PostgreSQL Case Sensitivity

PostgreSQL stores column and table names in lower case. JDBC queries on PostgreSQL retrieve column or table names in lowercase unless the names are quoted. For example:

```
use aliases Select m.ID AS \"ID\" from Department m
```

Use the backslash as an escape character in the class file, but not in the `persistence.xml` file.

Sybase Finder Limitation

If a finder method with an input greater than 255 characters is executed and the primary key column is mapped to a VARCHAR column, Sybase attempts to convert type VARCHAR to type TEXT and generates the following error:

```
com.sybase.jdbc2.jdbc.SybSQLException: Implicit conversion from datatype
'TEXT' to 'VARCHAR' is not allowed.  Use the CONVERT function to run this
query.
```

To avoid this error, make sure the finder method input is less than 255 characters.

MySQL Database Restrictions

The following restrictions apply when you use a MySQL database with the GlassFish Server for persistence.

- MySQL treats `int1` and `int2` as reserved words. If you want to define `int1` and `int2` as fields in your table, use ‘`int1`‘ and ‘`int2`‘ field names in your SQL file.
- When `VARCHAR` fields get truncated, a warning is displayed instead of an error. To get an error message, start the MySQL database in strict SQL mode.
- The order of fields in a foreign key index must match the order in the explicitly created index on the primary table.
- The `CREATE TABLE` syntax in the SQL file must end with the following line.

  ```
  )  Engine=InnoDB;
  ```

  `InnoDB` provides MySQL with a transaction-safe (ACID compliant) storage engine having commit, rollback, and crash recovery capabilities.
- For a `FLOAT` type field, the correct precision must be defined. By default, MySQL uses four bytes to store a `FLOAT` type that does not have an explicit precision definition. For example, this causes a number such as 12345.67890123 to be rounded off to 12345.7 during an `INSERT`. To prevent this, specify `FLOAT(10,2)` in the DDL file, which forces the database to use an eight-byte double-precision column. For more information, see `http://dev.mysql.com/doc/mysql/en/numeric-types.html`.
- To use `||` as the string concatenation symbol, start the MySQL server with the `--sql-mode="PIPES_AS_CONCAT"` option. For more information, see `http://dev.mysql.com/doc/refman/5.0/en/server-sql-mode.html` and `http://dev.mysql.com/doc/mysql/en/ansi-mode.html`.
- MySQL always starts a new connection when `autoCommit==true` is set. This ensures that each SQL statement forms a single transaction on its own. If you try to rollback or commit an SQL statement, you get an error message.

  ```
  javax.transaction.SystemException: java.sql.SQLException:
  Can’t call rollback when autocommit=true
  ```

  ```
  javax.transaction.SystemException: java.sql.SQLException:
  Error open transaction is not closed
  ```

  To resolve this issue, add `relaxAutoCommit=true` to the JDBC URL. For more information, see `http://forums.mysql.com/read.php?39,31326,31404`.
- MySQL does not allow a `DELETE` on a row that contains a reference to itself. Here is an example that illustrates the issue.

  ```
  create table EMPLOYEE (
          empId   int         NOT NULL,
  ```

```
        salary  float(25,2) NULL,
        mgrId   int         NULL,
        PRIMARY KEY (empId),
        FOREIGN KEY (mgrId) REFERENCES EMPLOYEE (empId)
        ) ENGINE=InnoDB;

        insert into Employee values (1, 1234.34, 1);
        delete from Employee where empId = 1;
```

This example fails with the following error message.

```
ERROR 1217 (23000): Cannot delete or update a parent row:
a foreign key constraint fails
```

To resolve this issue, change the table creation script to the following:

```
create table EMPLOYEE (
        empId   int         NOT NULL,
        salary  float(25,2) NULL,
        mgrId   int         NULL,
        PRIMARY KEY (empId),
        FOREIGN KEY (mgrId) REFERENCES EMPLOYEE (empId)
        ON DELETE SET NULL
        ) ENGINE=InnoDB;

        insert into Employee values (1, 1234.34, 1);
        delete from Employee where empId = 1;
```

This can be done only if the foreign key field is allowed to be null. For more information, see http://bugs.mysql.com/bug.php?id=12449 and http://dev.mysql.com/doc/mysql/en/innodb-foreign-key-constraints.html.